Computing Project
UFCE3B-40-3
Project Draft

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

April 7, 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 observed that the number of components which can be deployed onto a semiconductor integrated circuit, or 'chip' appeared to be exponentially doubling every year (Mack, 2011). Moore conjectured that this trend would continue for the proceeding ten years; with increasing chip area, decreasing feature size and improved circuit designs (Mack, 2003).

Moore did not however, observe that such rapid growth is infinitely sustainable, and predicted a decline 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 environment 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 addressed the AFIPS Spring Joint Computer Conference (Amdahl, 1967). He argued 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, almost 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 interestingly, a paradigm which may facilitate this brave new world of parallel development was first published in the 1930's, implemented in the 1950's and is subject to further refinement (Church, 1936; McCarthy, 1960; Fogus & Houser, 2011). It is known as functional programming, and could potentially aid in the production of safe concurrent software (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) postulated 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

To what extent can functional programming exploit the potential performance gains of multi-core processors?

2 Background

In order to fully grasp the implications of utilising a functional language, it would first serve to define in as exact terms as possible what a functional language is, and what distinguishes one from its imperative counterparts.

To this end, the reader is presented with a simplified introduction to the mathematical model upon which functional programming is based. Following this, and with the intention of further crystallizing the definition of what it means to program in a functional manner, the first true functional language LISP is presented. Finally for functional programming, the language conceived with the express purpose of approaching concurrent programming is presented; the reader learns about Clojure.

With a newfound understanding of functional programming, it is now that the challenges of writing parallel programs are examined alongside the current state of the art industry best practices, which attempt to manage concurrency and stop it getting out of hand.

Finally, the suitability of applying Clojure to the aforementioned challenges is investigated.

2.1 Programming Languages

2.1.1 Imperative Programming

The overwhelming majority of programs in execution today are written in imperative programming languages. An imperative language can be said to have had its method of computation derived from Alan Turing's Universal Turing Machine (Turing, 1936; Barendregt, Manzonetto & Plasmeijr, 2013); and has been loosely defined as a language *"where a sequence of statements mutates program state"* (Fogus & Houser, 2011). It is assumed that the reader of this document is experienced with at least one imperative language such as Java, C, Ruby, Python. . . etc. As such, it is determined that further explanation of the imperative paradigm is unnecessary, as it is well understood by the computing community in general.

2.1.2 Functional Programming

Fogus & Houser postulate that functional programming is a difficult term to accurately define, as any given definition will be biased towards the specific functional language used by the individual in question. They continue, stating that even within academia it can often be contradicted from one paper to the next (Fogus & Houser, 2011). For a field in which precise definitions are foundational; an inconsistent view of what constitutes functional programming is a formidable issue to be sure. Fortunately however, Fogus & Houser provide a definition which covers the core tenets of functional programming regardless of language or viewpoint:

> "Functional programming concerns and facilitates the application and composition of functions. Further, for a language to be considered functional, its notion of function must be *first class*. The functions of a language must be able to be stored, passed and returned just like any other piece of data within that language." (Fogus & Houser, 2011)

With this definition as a starting point it is possible to see the how the LISP family of functional languages were derived from Alonso Church's untyped lambda (λ) calculus (Church, 1936; McCarthy, 1960). It would therefore be beneficial to first discuss and understand some basic principals of λ calculus before considering the features of specific functional languages.

2.1.3 λ

In 1932 Alonso Church published 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 was created with the intent to allow the proof of computations by methods of substitution and variable binding (Rosser, 1984). Unfortunately, the paper and the method within were proved inconsistent when subjected to further investigation (Rosser, 1984).

Unperturbed, Church returned in 1936 with a paper proving the impossibility of the Entscheidungsproblem (Church, 1936; Copeland, 2004). He achieved this using a refined and much stronger model of computation; the untyped λ calculus (Church, 1936). Independent of Church's work, Alan Turing also addressed the Entscheidungsproblem, 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).

2.1.4 λ Notation & Evaluation

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).

$$
\begin{aligned}
< expression > \; &:= \; < name > \mid < function > \mid < application > \\
< function > \; &:= \lambda < name > . < expression > \\
< application > \; &:= \; < expression >< expression >
\end{aligned}
$$

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 to and returned from other functions (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.

The binding of variables to the λ operator introduces an important functional aspect known as a closure; in which any functions nested within the body of another function will use the parent function's bound variable. For example:

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

Here we have a function which takes the far right argument x and binds it to λx. The body of λx consists of another function which takes the value of y, binds it to λy and then returns the value of applying x to y. Due to the closure, the value of x in λy is the same as that bound to λx.

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 a variable or value. Currying is made possible by the use of closures, as bound variables remain bound even within nested functions which are returned. If a function definition which accepts three variables x, y, z and returns another variable r is considered; 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 (arguments):

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

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 a function which adds two numbers together, supplied with the arguments 5 and 6:

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

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

$$(\lambda x.x)5$$

Which reduces to:

$$5$$

More complicated functions can be reduced in a series of steps:

$$\begin{aligned}
&(((\lambda x.\lambda y.\lambda z.(x+y+z))3)4)5\\
&((\lambda y.\lambda z.(3+y+z)4)5\\
&(\lambda z.(3+4+z)5\\
&12
\end{aligned}$$

λ calculus introduces some important concepts to the field. The notion of using the substitution method through pure functions which have no side effects[1], allows for the definition of programs which can be completely deterministic and mathematically tractable. Because of this tractability, functions can safely be treated as first class citizens, used as arguments and returned from other functions. If Fogus & Howser's definition of functional programming is considered, then λ calculus would indeed appear to be genesis.

[1]Functions do not modify any state and are consistent when given the same inputs (Banning, 1979).

2.1.5 LISP

List Processing, or LISP was developed by John McCarthy between 1956 and 1960. LISP was born from the fundamentals of λ calculus and the first attempt at an artificial intelligence centric language called *Information Processing Language* (McCarthy, 1960; McCarthy, 1981). It is considered high level and is the first programming language to implement garbage collection, recursion and a complete if-then-else statement (McCarthy, 1960; McCarthy, 1981; Graham, 2003).

Its form is instantly recognisable from the use of parenthesis to define expressions *e.g. Code 1* and its use of polish prefix notation (McCarthy, 1981) in which the operand or function is applied to all proceeding elements; as in the example *Code 2* from Abelson & Sussman (Abelson & Sussman, 1996).

```
(+ (* 3
      (+ (* 2 4)
         (+ 3 5)))
   (+ (- 10 7)
      6))
```

Code 1: Parenthesised LISP with prefixed operands

```
(+ 21 35 12 7)
= 75

(* 25 4 12)
= 1200
```

Code 2: Polish prefix notation resolution

All data within a LISP program is either of two data types, an atom or a list. A list is a sequence of elements, each element of which is itself either an atom or a list. Whereas an atom is either a number or a symbol. Lists themselves consist of primitive pairs that act as building blocks for more complex structures (McCarthy, 1960; Abelson & Sussman, 1996).

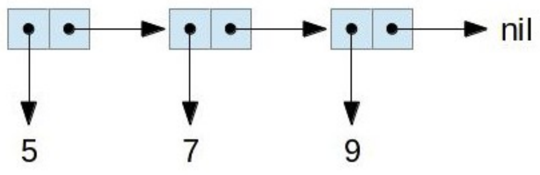

Figure 1: Lists and Atoms

The above list in *Figure 1* is represented in *“box-and-pointer notation”* (Abelson & Sussman, 1996) and would evaluate to a list of the numbers: $[5, 7, 9]$. Each element consists of two pointers, which point to either an atom or a list. It is perhaps not immediately apparent, however using this simple method of structuring data it is possible to model complex items *Figure 2*.

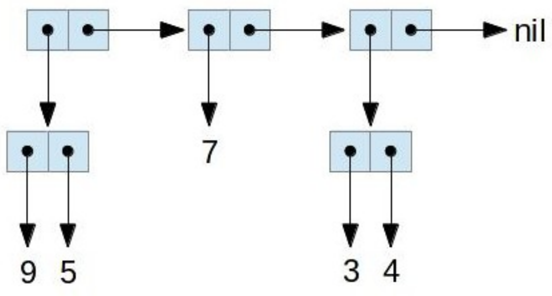

Figure 2: A more complex list structure

A LISP program consists of symbolic expressions (or S-expressions). These expressions are evaluated and converted into the aforementioned list structures which then reside in memory (McCarthy, 1960; McCarthy, 1981). Thus the textual code written in the example *Code 3* is interpreted into an actual data structure shown in *Figure 3* (Abelson & Sussman, 1996).

```
(+ (* 3 3) 4)
```

Code 3: LISP as written in editor

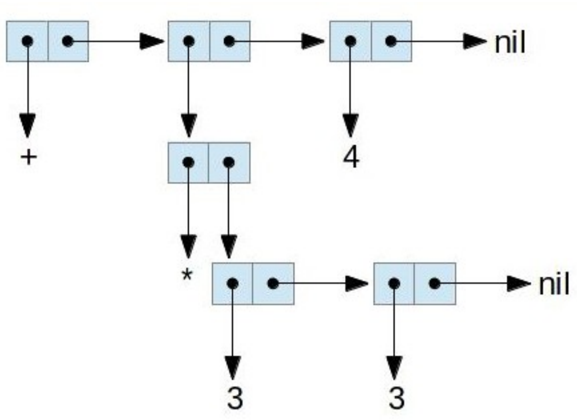

Figure 3: LISP code as a data structure

The inherent power of this method of interpretation is that the code written in the text editor now exists in a form which can be manipulated at run time (McCarthy, 1960). Code is stored in the same manner as data, is just as malleable and can be passed around the program in the same way. This allows the LISP programmer to utilize a powerful language construct called a macro to create, as George Graham stated *"programs that write programs"* (Graham, 2003).

The original LISP specification consists of a mere seven functions and two special forms. Like the λ calculus it manages to describe *"the whole of computation"* (Fogus & Houser, 2011) using a minimalistic and elegant syntax (McCarthy, 1960). It is considered by some to still be amongst the most the most powerful languages available, regardless of it's lack of widespread commercial use (Graham, 2003; Fogus & Houser, 2011).

2.1.6 Clojure

Clojure is a response to a perceived need. The

2.2 Concurrency

Con · cur · rent

1. Occurring or existing simultaneously or side by side . . .

(Dictionary.com, 2012)

Concurrency effectively means that things happen at the same time. When speaking of concurrency in relation to software development, one of two processes are occurring:

1. **Simulated concurrency:** The host platform has a single processor; therefore threads and processes are interleaved and are scheduled to use the processor at different intervals. This affords an illusion of simultaneous execution.
2. **Simultaneous execution:** The host platform has multiple processors or processor cores and can execute threads and processes simultaneously.

Regardless of the process which is in occurrence, once a developer decides to use threads or multiple processes, the net result is the same and concurrency becomes a very real issue and concern. The prime directive of the concurrent developer is to not allow the system to produce inconsistent data; that is to ensure that all related data (Goetz, Peierls, Bloch, Bowbeer, Holmes & Lea, 2006).

2.2.1 State and Mutability

Any discussion regarding the issues related to concurrency almost inevitably leads to the consideration of both state and mutability.

> “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, class 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)

If then, a root cause of concurrent programming challenges is sought, the industry need look no further than change over time, or mutability. Rich Hickey postulates that when developing using mutable objects, the state of the resulting system becomes difficult to both predict, replicate and test (Hickey, 2009). The issue is further compounded when interconnected mutable objects are introduced to multiple threads and the developer must now deal with the inherent issue of non-determinism which concurrency introduces (Sutter & Larus, 2005).

The issue of controlling the identity of an object over time is not something revelatory, however something which is currently accepted and understood as a challenge of computer science. It is however an issue which has been brought to the fore now that processing power is becoming increasingly parallelized (Sutter & Larus, 2005; Hwu, Keutzer & Mattson, 2008).

2.2.2 Race Conditions

Consider the flow of execution in *Figure 4*. Two threads both wish to increment a

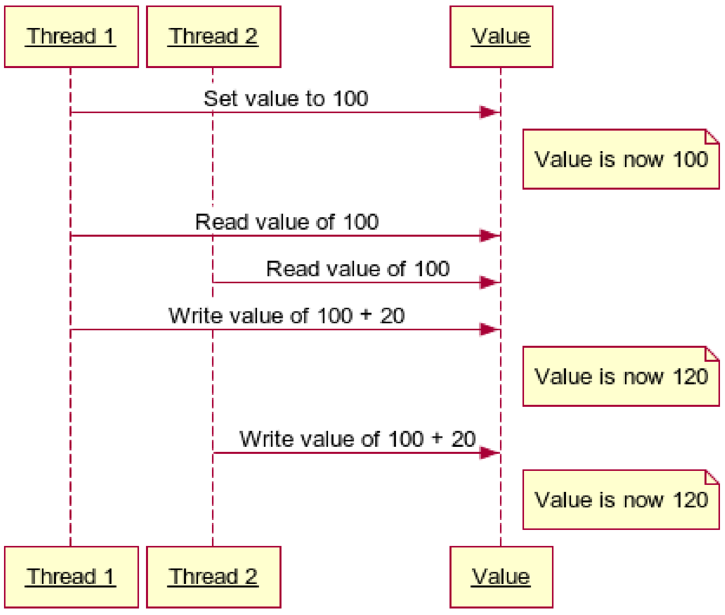

Figure 4: Race conditions on mutable data

mutable variable by 20 each. The variable in question is initially set to have a value of 100. The expectation therefore is that the variable's value will be 140 when both threads have finished. Unfortunately, as threads introduce a non-deterministic outcome to a program's execution it's entirely possible that the outcome will be that of *Figure 4*; whereby both threads read a value of 100, rather than the first thread reading a value of 100 and the second reading a value of 120. The result of these operations is inconsistent and makes the system in which it resides completely probabilistic and thus difficult to understand.

The nature of concurrent execution means that the developer can never be sure which thread of execution will get to a variable first, or indeed if a thread will be able to both read and write the data before another thread also reads and writes the data. These *race conditions* require a good strategy in order to be well marshalled.

2.2.3 Deadlock

Deadlock occurs when two or more threads obtain exclusive locks for specific shared resources and then each require access to the resources the other has locked.

In *Figure 5* it is demonstrated that thread 1 gains access to resource A and thread 2 gains access to resource B. In order to continue operation, thread 1 also requires resource B and thread 2 requires resource A. With both resources already being locked, both threads will sit in limbo awaiting the release of a resource which will never happen. Both threads are blocked and now completely useless.

2.3 Concurrent Strategies

In order to effectively control mutability, the industry has developed several strategies which allow the management of concurrently accessible mutable data (Goetz

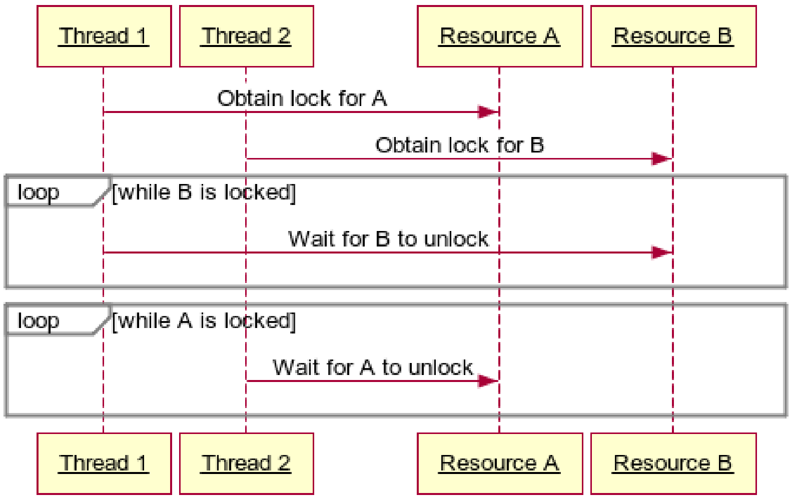

Figure 5: Deadlock of two threads

et al., 2006). Unfortunately, even the best sources of concurrent strategies concede that this is an area which is very difficult to get right, and that if data were immutable the development of concurrent systems would be much easier (Goetz et al., 2006; Hwu et al., 2008). That being said, let's examine some current best practices for concurrent software development.

2.3.1 Locking

If a developer requires that a resource needs to be safe from inconsistent reads and writes they can lock the aforementioned area. This allows a single thread to obtain the key to the resource and gain exclusive access to that particular piece of code. The side effect of locking is that all other threads which also need access to the locked area are now blocked and must wait until they have sole ownership of the key before proceeding (Goetz et al., 2006).

This appears to be a sound idea in principal; however there are some issues with locks; especially for multiple stage operations.

Firstly, locking is a convention or an agreement, which is adhered to by software developers and is not typically enforced by any mechanistic means. This becomes a large issue, as not every resource a developer uses needs to be locked. It introduces a fallible component (namely a human) into the system, as the developer must keep track of which resources need to be locked and which can be used without locks. Additionally, should a decision be made to lock a previously unlocked resource, all locks must be retrofitted to existing code, again by fallible human means.

Locks also introduce bottlenecks into the system. As one thread gains access to a particular resource all other threads requiring the resource are now blocked, unable to continue their execution until such a time that they can get hold of the key. This is an issue on single core machines, but on a multi-core machine the net result is a massive waste of processing power (Hwu et al., 2008).

Before continuing it would be useful to define two groups which wish to access shared resources: readers and writers. Readers, as you would expect require read only access to a particular resource and will not introduce inconsistent data to a system, whereas writers require both read and write access and could potentially cause data integrity issues if left unchecked.

If a typical lock is considered, then when a reader has a key it blocks all other readers from accessing the same resource. This is blatantly inefficient as locks exist to protect data integrity from writes. Therefore readers need not block other readers from accessing a resource which they are not going to change (Goetz et al., 2006).

The solution to this particular issue is the use of a more complex lock which allows multiple readers to access a resource, but only a single writer. This is a much improved strategy, however still introduces the blocking aspect during writes.

There are two basic locking strategies which can be used, each with their own trade offs and considerations.

2.3.2 Coarse Granularity Locking

In terms of parallel programming complexity managing a single resource with locks is child's play. Unfortunately, in the real world operations are performed on multiple resources by multiple threads and can require several locks at any given time. This introduces deadlock as previously described (Goetz et al., 2006).

A strategy to deal with this is coarse granularity locking. This implies that a number of resources are obtained for a thread's exclusive use with a single lock. This is simple to both describe and document, and is considered to be one of the safest locking schemes for multiple resources (providing that each resource only appears in a single lock).

Naturally, there is a trade off for such ease of use, namely throughput. Coarse granularity locking instantly makes any resources included in the lock blocking whether the thread which has obtained the lock actually requires them or not.

2.3.3 Fine Granularity Locking

If throughput is of concern when locking resources then fine granularity locking is an effective strategy to utilize. With fine granularity, each critical resource has its own lock. The key to a workable fine granularity locking is to effectively determine a locking strategy which all developers can follow. Failure to do so will likely result in deadlock (Goetz et al., 2006).

Achieving a workable and consistent locking strategy for the order of obtaining locks is difficult, as all systems are subject to changes. New resources will be added, new locks implemented and the developers must spend considerable effort to maintain the integrity of the locking order.

2.3.4 Copy On Write Collections

Useful for when multiple threads are operating on a collection. Within a copy on write collection, if any changes are made to the data within, a copy of the collection is made. This copy is then atomically swapped for the previous version of the collection in a single operation. The net effect of this is that any reads of the collection will result in the reader being presented with a copy of the collection data that will never change. This allows the use of collections which are completely non-blocking (Goetz et al., 2006).

Unfortunately, copy on write collections still have their limitations. In terms of performance, the write operation is computationally expensive as the collection data is being copied in it's entirety. Copy on write collections perform admirably when used mainly for reading with occasional writes; however do not stand up well to excessive write operations (Goetz et al., 2006).

Furthermore, operations comprised of multiple steps still require a lock, as copy on write still allows for race conditions if multiple read/write steps are required from separate concurrent operations (Goetz et al., 2006).

2.4 Clojure's Concurrent features

— 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

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

2.5 GPGPU

When speaking of processors and processing throughput it is becoming somewhat impossible to ignore the impact of GPGPU upon massively parallelized algorithms (Poulding, 2012).

2.5.1 What is GPGPU

GPGPU, short for general-purpose computing on graphics processing units is the act of utilizing the processing unit of a computer's graphics card in order to perform computations. Modern graphics cards have a many core architecture comprised of several hundred small processing cores which share the graphic card's memory (Poulding, 2012).

When speaking of GPGPU, most sources refer to nVidia's CUDA library and architecture which is exposed through either the use of native C code or via a series of libraries which are available in Java, C# and other mainstream programming languages (Poulding, 2012).

GPGPU is currently subject to widespread interest and applicative research, especially from computer scientists focused on search (Risco-Martin, Lanchares & Coello-Coello, 2012)

2.5.2 GPGPU Benefits

In short, GPGPU results in massive throughput, providing that Amdahl's Law (Amdahl, 1967) is circumvented by a well parallelized algorithm.

In short, and providing that Amdahl's Law is circumvented by a well parallelized algorithm (Amdahl, 1967); massive throughput. Speed increases of up to

25x have been recorded on GPGPU when compared to a more traditional processor configuration (Yoo, Harman & Ur, 2011).

2.5.3 GPGPU Issues

At the moment, the main issue when considering GPGPU as an option for improving throughput is that of memory latency (Poulding, 2012). As the GPU cannot interface with the main system memory or processor cache, all data must be copied to the GPU's limited internal memory, an expensive operation which wastes clock cycles if performed needlessly (Poulding, 2012).

Perhaps contrary to most other studies performed, Darren Chitty has demonstrated that when cache memory is utilised correctly on multi-core processors, they can garner massive gains in throughput and in some cases outperform the GPU (Chitty, 2012).

2.5.4 GPGPU Summary

It appears that the current GPU has an architecture similar to the proposed many-core processors (Borkar, 2007; Poulding, 2012). As such, they offer incredible throughput (Yoo et al., 2011), but offer much more pedestrian memory access speeds (Chitty, 2012). That being said, they're subject to much current interest and provide an interesting alternative to programming for current CPU's (Risco-Martin et al., 2012).

2.6 Continuing research

put links to journals on high speed research here

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 break down a software problem into much smaller component parts than would otherwise be possible.

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. It is also the expectation that concurrent programming tasks will take less time to implement, as less time should be spent attempting to debug threaded code.

4 Objectives

This section details the main objectives of the paper along with a list of potential metrics which could be measured implemented in the software product. Also discussed are platform constraints and the extent to which the performance of a platform relates to the performance of a given language.

4.1 Objectives

This project sets out to determine if Clojure can to outperform traditional imperative languages and GPGPU when faced with a suitably concurrent problem. To this end, the same case study will be implemented in a number of different languages, whereupon the relative performance of the languages in question will be compared. The following metrics could be measured.

1. Execution time
2. Consistency
3. Memory usage
4. 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.

4.2 Platform and language

When considering the execution speed of a language, it is likely worth considering the impact which the language platform has on the relative throughput. For instance, Clojure operates atop the Java virtual machine and has many of its features implemented in Java. This adds a level of ambiguity when considering performance; should it be attributed to the language features of Clojure, or to the cleverness of the Java Virtual Machine?

In order to address this ambiguity, it would be useful to include Java in a control group, thus allowing a comparison of pure language features.

5 Requirements Analysis

To ensure that the objectives of this paper are satisfied, a list of potential features are be drawn up and subsequently prioritised. Following this, a test approach and methodology by which to exercise the test approach are formulated.

Finally, the implementation of the control group is determined along with a suite of acceptance tests

5.1 Feature List

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 a list of features which would satisfy the defined objectives (Clegg, 1994).

5.1.1 Must

- **Multi-Threaded:** The application uses threads in order to take advantage of multiple CPUS.
- **Shared Data Reads:** The application utilises shared data which is read from by multiple threads.
- **Shared Data Writes:** The application utilises shared data which is written to by multiple threads.
- **Execution Time Logging:** The application can log it's overall execution time and the execution time of individual threads.

- **Implementation in Clojure:** The application will be implemented using the Clojure programming language.
- **Implementation in Java:** The application will be implemented using the Java programming language.

5.1.2 Should

- **Consistency Checking of Shared Data Access:** The application can check its shared data for instances of inconsistent reads/writes.
- **Test Suite:** The application is validated by a suite of unit tests in order to confirm that it operates correctly.
- **Implementation in CUDA:** The application will be implemented using the C++ programming language and the GPGPU CUDA library.

5.1.3 Could

- **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.
- **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.

5.1.4 Wont

- **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.
- **Implementation in Python:** The application will be implemented using the Python programming language.
- **Implementation in Ruby:** The application will be implemented using the Ruby programming language.

5.2 Test Approach

5.2.1 Control Group

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

1. **Execution Time :** How quickly an application completes..
2. **Consistency :** The ability of the system to not produce errors when multiple threads are accessing shared data.
3. **Production Time :** the length of time taken to implement the application.

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

1. **C++ CUDA :** 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. Included because it is the current state of the art and has achieved exceptional performance benchmarks in other studies.

2. **Java :** The language that Clojure was originally written in and with which it shares a platform. Included because both languages run on the Java Virtual Machine, and thus that language features can be directly compared.

5.2.2 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 the test machine.

Execution Time The time taken 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 data, checking it's final state against a set pre-determined value. Any deviation from the predetermined value will confirm inconsistency. Deviations will be logged and subjected 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 have all recorded metrics analysed for their mean value, and standard deviation.

Test Machine The test machine for all runs will be an ASUS NV56-VZ laptop.

- **Processor:** i7 3610QM 4 cores@2.3ghz, hyper-threaded up to 8 cores, turbo boost up to 3.3ghz.
- **Memory:** 8gb,
- **Graphics processor:** nVidia GT650m, 394 cores@900mhz, 2gb video ram.

5.3 The Travelling Salesman Problem

In order to successfully satisfy the must have requirements of the control group; a problem is required that has a finite amount of associated computation, a definite solution, shared data and can be parallelized. Search Based Software Engineering is rife with examples of problems which can be effectively parallelized (Yoo et al., 2011); few of which are as famous as the travelling salesman problem (Johnson & McGeoch, 1997).

5.3.1 Problem Definition

The travelling salesman problem presents a search question. Given a set of cities and a distance between each pairing of cities within the set, what is the shortest tour length? Where a tour consists of visiting each city exactly once (Johnson &

McGeoch, 1997). This problem can be visualised, as in *Figure 6*, which consists of four cities.

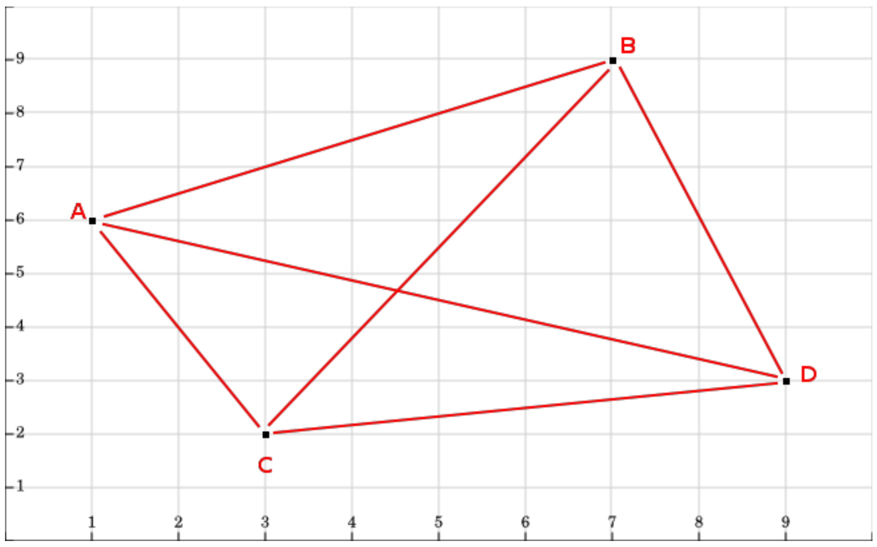

Figure 6: Cities on a plane, displaying routes between pairs

The travelling salesman problem is interesting in that the number of possible tours is the factorial of the size of the set of cities. For instance the example in *Figure 6* has 24 different tours. If a set of 12 cities is used there would be 479,001,600 tours (Johnson & McGeoch, 1997).

This presents a problem with a massive search space and a well defined solution, that being the shortest tour. It also presents an opportunity to utilise a parallel algorithm which concurrently calculates the length of a tour; and determines if it is currently the shortest tour evaluated.

5.3.2 Search Method

With the travelling salesman being a search problem, there are a myriad of strategies which can be utilised in order to determine the shortest tour in the shortest possible amount of time (Johnson & McGeoch, 1997).

In a test of raw language throughput however, the use of advanced searching algorithms and techniques would be superfluous at best. Therefore the computer programs produced will use a brute force approach; examining every possible tour and determining which is the shortest (Schaeffer, Lu, Szafron & Lake, 1993).

5.3.3 Acceptance Tests

In order to ensure that the programs produced correctly evaluate which tour is shortest for any given set of cities, a number of test cases with known results have been formulated.

Test cases are presented as a list of x and y co-ordinates as follows:

$$[[1x, 1y], [2x, 2y], [3x, 3y] \ldots [nx, ny]]$$

1. $[[1, 1], [4, 4], [7, 7], [10, 10]]$

   Shortest tour distance: 12.7279

   Tests that distances are calculated correctly when cities do not share either x or y co-ordinates.

2. $[[1, 1], [1, 4], [5, 5], [1, 7]]$

   Shortest tour distance: 10.4721

   Tests that distances are calculated correctly when two or more cities share an x co-ordinate.

3. $[[1, 1], [4, 1], [6, 6], [9, 1]]$

   Shortest tour distance: 13.8309

Tests that distances are calculated correctly when two or more cities share an y co-ordinate.

4. $[[1, 1], [1, 2], [1, 5], [3, 5], [7, 5]]$

   Shortest tour distance: 10.0

   Tests that distances are calculated correctly when multiple cities share either an x or y co-ordinate.

5. $[[1, 1], [4, 4], [7, 7], [10, 10], [3, 3], [6, 6], [9, 9], [2, 2], [5, 5], [8, 8], [8, 9], [5, 7]]$

   Shortest tour distance: 15.3137

   A test with 12 cities and 479,001,600 possible tours. A stress test to ensure that the programs can cope with a large search space.

It is expected that the aforementioned test cases will sufficiently exercise complete operation of the programs produced and ensure that results gathered are both complete and correct.

6 Design

6.1 Architecture

The design of an application to satisfy the objectives of this paper will be completed in three stages.

1. Firstly, a decision must be made regarding what is actually to be built. The objectives and features which must be satisfied have been well defined; however a question still remains regarding what will satisfy the test specification.
2. A set of tests will be developed in order to ensure that the computer program correctly addresses point 1. There must be no doubt that the program can successfully solve the problem in question.
3. The computer program itself will be designed and implemented with rapid prototyping and iterative development.

6.2 Components

Regardless of the language of implementation, a computer program designed to solve the travelling salesman problem will have distinct components. Hence the design of the program will attempt to remain as language agnostic as possible, with specific implementation details to be crystalized via rapid-prototyping. Individual components are discussed below, with a complete view of how they fit together displayed in *Figure 7*

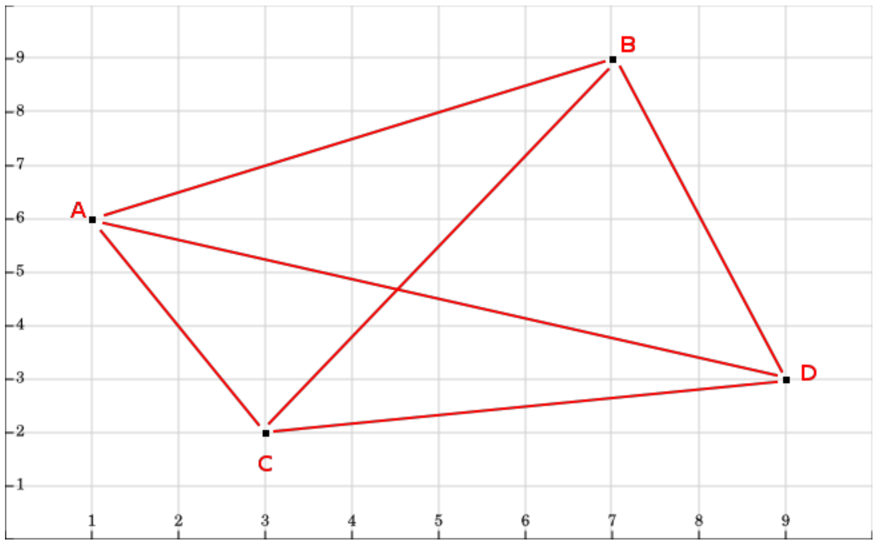

Figure 7: Interaction between program components

6.2.1 Run Controller

The run controller is responsible for orchastrating the entire program. It is the primary controller and utilises all other components. Crucially, it is also responsible for recording the execution time of any given run. It should accept command line arguments regarding which set of cities to utilise, as well as how many runs are required.

6.2.2 File Reader

The file reader is responsible for taking a file of the form: x,y:x,y:x,y:x,y and converting it to a set of cities which the rest of the program can use to calculate the results with.

6.2.3 Permutation Calculator

Each individual tour is an ordered permutation of a set of cities. In order to evaluate all possible tours, there needs to be a facility by which a specific permutation can be generated.

6.2.4 Distance Calculator

Once a permutation has been obtained, its total distance must be calculated.

6.2.5 Results Collector

The results of each run, the time and shortest distance will be recorded in a data structure or structures, which can later be subjected to statistical analysis.

6.2.6 Statistics Calculator

This will contain a series of methods which are able to calculate both averages and standard deviations for the collected results.

6.2.7 Results Printer

Some mechanism is required to print the results to the screen or perhaps an output file.

6.3 Flow of Control

7 Results

8 Conclusion

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.

Banning, J. P. (1979). An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables, *Proceedings of the 6th ACM SIGACT - SIGPLAN Symposium on Principles of Programming Languages*, ACM, pp. 29–42.

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.

Chitty, D. (2012). Fast Parallel Genetic Programming: Multi-core CPU Versus Many-core GPU, *Soft Computing* **16**(10).

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.

Dictionary.com (2012). Definition of concurrent, `http://dictionary.reference.com/browse/concurrent?s=t`: [Online; accessed 26-October-2012].

Fogus, M. & Houser, C. (2011). *The Joy of Clojure*, Manning Publications Co.

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D. & Lea, D. (2006). *Java Concurrency In Practice*, Addison-Wesley.

Graham, P. (2003). Beating the Averages, `http://www.gotw.ca/publications/concurrency-ddj.htm`: [Online; accessed 27-October-2012].

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

Hickey, R. (2009). Are We There Yet, `http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey`: [Online; accessed 15-October-2012]. JVM Languages Summit.

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.

Johnson, D. S. & McGeoch, L. A. (1997). The Travelling Salesman Problem: A Case Study in Local Optimization, *Local Search in Combinatorial Optimization* pp. 215–310.

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. (1960). Recursive functions of symbolic expressions and their computation by machine, Part 1, *Communications of the ACM* **3**(4): 184–195.

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.

Poulding, S. (2012). Tutorial: High performance SBSE Using Commodity Graphics Cards, *SSBSE 2012*, Springer, p. 29.

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

Risco-Martin, J. L., Lanchares, J. & Coello-Coello, C. A. (eds) (2012). *Soft Computing Special Issue on Evolutionary Computation on General Purpose Graphics Processing Units*, Vol. 16(10), Springer.

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

Schaeffer, J., Lu, P., Szafron, D. & Lake, R. (1993). A Re-Examination of Brute-Force Search, *Proceedings of the AAAI Fall Symposium on Games: Planning and Learning*, pp. 51–58.

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.

Yoo, S., Harman, M. & Ur, S. (2011). Highly Scalable Multi Objective Test Suite Minimisation Using Graphics Cards, *SSBSE 2011*, Springer, pp. 219–236.