Fall 12-1-2010

An evaluation of Go and Clojure

Robert Derek Stimpfiling
University of Colorado Boulder

An evaluation of Go and Clojure

A thesis submitted in partial satisfaction of the

requirements for the degree Bachelors of Science in

Computer Science

Fall 2010

Robert Stimpfling
Department of Computer Science
University of Colorado, Boulder

Advisor:

Kenneth M. Anderson, PhD
Department of Computer Science
University of Colorado, Boulder

1. Introduction

Concurrent programming languages are not new, but they have been getting a lot of attention more recently due to their potential with multiple processors. Processors have gone from growing exponentially in terms of speed, to growing in terms of quantity. This means processes that are completely serial in execution will soon be seeing a plateau in performance gains since they can only rely on one processor.

A popular approach to using these extra processors is to make programs multi-threaded. The threads can execute in parallel and use shared memory to speed up execution times. These multithreaded processes can significantly speed up performance, as long as the number of dependencies remains low. Amdahl's law states that these performance gains can only be relative to the amount of processing that can be parallelized [1]. However, the performance gains are significant enough to be looked into.

These gains not only come from the processing being divvied up into sections that run in parallel, but from the inherent gains from sharing memory and data structures. Passing new threads a copy of a data structure can be demanding on the processor because it requires the processor to delve into memory and make an exact copy in a new location in memory. Indeed some studies have shown that the problem with optimizing concurrent threads is not in utilizing the processors optimally, but in the need for technical improvements in memory performance [2]. Accessing and copying memory is painstaking compared to being able to just pass a reference to a memory structure to a new thread. However, this is also where most of the problems come from in multithreaded processes. For any thread to access shared memory, it must be sure that it is the only one modifying the memory.

The traditional way of protecting memory has been with mutual exclusion (mutex) locks that must be implemented by the programmer. While these solutions are widely used, they are not without their flaws. There is the famous "Dining Philosophers" problem, and quite a few others, where each thread is prevented from accessing resources it needs because of a different thread. Of course, there are plenty of algorithms and state diagram theories to deal with deadlock, but they seem to make a complicated problem even more complex complicated.

One of the solutions to making concurrent threads has been to add a library to an already popular programming language. This has the advantage of not having to learn or create an entirely new programming language that will compile on an unfamiliar platform. One example of this is the Java concurrency API. By including the java.util.concurrent packages, the programmer can use some basic concurrency tools, such as locks, immutable objects, joins, etc [3]. While this support is useful, it still forces the programmer to manually manage memory and deal with deadlocks.

Although it can be argued that these solutions are simple enough for the average programmer to understand and implement, there are some alternatives that not only make concurrent programming simpler to the programmer, but also let the programmer create powerful programs without having to worry about locks. There are some programming language designers who made concurrent programming simple and more accessible to the common programmer. In fact, within the past year, two major programming languages have been released, both of which

boast powerful concurrency features. Clojure 1.0 was released in mid 2009 by Rich Hickey, and Go was released in late 2009 by Google. Specifically, Go was designed primarily by Robert Griesemer, Ken Thompson, and Rob Pike. Both of these languages seek to make concurrency a more user-friendly feature, free from locks that the programmer has to implement manually.

Of course, all of these concurrency features do not mean very much to a programmer unless the language itself is also appealing. That is to say, a programming language whose only appeal is a better concurrency model is not very appealing at all. Anyone who is comfortable with the standby languages such as C, C++, Common Lisp, Java, etcetera, has little incentive to switch to a new language, short of something revolutionary. Each standby language has its problems, but they are adequate enough to not warrant any massive landscape changes in the programming community every time a new language comes out. To be worthwhile, Go and Clojure attempt to address more than just an outdated concurrency model.

1.1 Clojure: Background and appeal

Available on the Clojure Google group, Rich Hickey's "Are We There Yet?" presentation attempts to explain why something new is needed [9]. In it, Hickey questions whether or not the popular object oriented languages are the way to go. He likens popular OO languages such as Smalltalk, Java, C#, and Python to different cars on the same road; while each has its significant differences, preferences between them are based more on programmer sensibilities than core principles.

Hickey's approach to advertising his new language wasn't only to address problems with OO languages, but to take a step away from OO protocols altogether. One of the major problems, according to Hickey, is that it is nearly impossible to determine the scope of effects changing a portion of the code will have. When a lot of references to memory are passed around, it's hard to tell what changing one function will do to the entire program. One of the staples of Clojure, "Pure Functions," addresses this problem. Pure functions are completely local, meaning there have no remote inputs or side effects. When a value is being modified by a Clojure function, it cannot be modified by any other function.

According to Hickey, the reason pure functions are superior is because they replicate the process of human vision. Humans associate an object with the image they have stored in their head. Modifying an object in real life creates a new image for one to associate it with. Creating a reference to memory is like taking a picture of an object; the object can go through many changes while the representation remains static. Hickey believes that when one creates a pointer or reference to memory, one conflates "symbolic reference with actual entities." The pointer becomes confused with the value it actually contains much like looking at a picture while modifying the actual object. Such an abstraction is not intuitive when changes occur frequently to the object.

However, pure functions are not always applicable, especially in situations where synchronizing data necessary. Instead of manipulating shared memory through the use of locks, Clojure uses the Software Transactional Memory (STM) system. Clojure implements this system through the use of Atoms, Agents, Refs, and Transactions. These data structures

Refs can be thought of as a reference to memory, somewhat like a pointer. However, refs are bound to a single location and can only be modified through the use of a transaction. When accessing the value stored by a ref, the thread is passed back a snapshot of the ref rather than the actual memory. When a ref is changed through a transaction it operates much like a database; the transaction updates the ref not by changing the value in memory, but by committing the ref to a new value. Agents are similar to Refs, but can be modified by passing an "action." An action is simply a function that is applied to the agent, and whose return value becomes the new agent.

Atoms are similar to agents, however they allow for synchronous changes when passed an action. If the atom is updated during an attempted action, the action is performed again with the updated atom in a spin loop. The action must be free of side effects, since it could be performed several times before it updates the atom.

In addition to addressing what Hickey believes to be inadequacies of Object Oriented languages, Clojure runs on the Java Virtual Machine. Hickey chose the JVM because he considers it to be an "industry standard, open platform" [4]. Running Clojure on a virtual machine definitely has its advantages since the JVM can run on virtually any operating system. This ensures that Clojure is accessible to nearly all programmers. There have been countless discussions on the performance of a program running on the Java Virtual Machine versus a program running directly on an operating system, but a lot of studies have shown that the performance is comparable [5]. Whatever the case, Hickey thought whatever potential performance losses, if any, due to running on the JVM were acceptable in order to have Clojure be compatible with such a popular platform. By running on the JVM, Clojure has the advantage of being able to use Java libraries. This is a huge advantage for anyone who has experience with Java and already has a preferred set of Java utilities or types.

1.2 Go: Background and Appeal

Unlike Clojure, Go is not trying to address the inherent flaws in Object Oriented programming languages. Instead, Go is a more of a refinement of system languages such as C and C++.

One of the main problems, according to Pike et. al, is the time it takes to compile code. Long compile times are hard to escape in what Pike calls, "a world of sprawling libraries." To reduce compile time, Go programs compile into "package files" which also contain transitive dependency information. Pike himself best describes this process in a Presentation given in July 2010:

> If A.go depends on B.go depends on C.go:
> - compile C.go, B.go, then A.go.
> - to recompile A.go, compiler reads B.o but not C.o.
> At scale, this can be a huge speedup

Indeed, in an impressive demo during a tech talk, Pike builds the complete Go source tree, which is around 130,000 lines of code, on his laptop. This process only takes 8 seconds [6].

Although Go benefits performance-wise from being a systems language, it currently has portability issues, as it is incompatible with the Windows platform. Go is only compatible with

UNIX based operating systems and Mac OS X at the time of this writing [7]. This comes with the territory of being a systems language, as compilers are difficult to standardize across all platforms. This is distinct from Clojure since Clojure's appeal is largely based on the fact that it can be run on any platform that has a JVM.

In direct contrast to Clojure's "No to OO" approach, Go was built to be an object-oriented language. However, it is unusual in the fact that there are no classes or subclasses. Instead, types can have methods, even basic types such as integers and strings. To satisfy a type, the value must have that type's interface. This means that since the empty interface has no methods, virtually every type can satisfy an instance that calls for an empty interface. [12]

Also unlike Clojure, Go allows the user to directly access and modify shared memory. In Go, maps and slices are reference types. Slices are representations of an array. A slice is passed an array, and all changes to the array are made through accessing the slice just like an array. The difference between a slice and an array is that slices can constitute any portion of an array, so a single array can be divvied up between several slices. Also, arrays are also pass-by-value, whereas slices of the array are pass-by-reference. In this way, an array can be modified by many different slices with no risk of corrupting the memory as long as the slices are partitioned correctly.

Something Go and Clojure do have in common is the fact that each language has its own set of utilities for concurrency and parallelization. Concurrency in Go is handled through "goroutines" and channels. In order to call a Goroutine, one simply needs to prefix a function call with the "go" keyword. Goroutines can be thought of roughly as threads. A function is passed to a goroutine, which executes the function in parallel to other goroutines. However, goroutines are multiplexed onto multiple OS threads, so if one goroutine blocks a resource, the other goroutines continue to run. When the goroutine exits, it exits silently. The documentation compares this effect to the UNIX '&' notation for running commands in the background.

Synchronizing goroutines is done through the use of channels. Channels constitute one of the three standard types that are reference types, maps, slices, and channels. They allow communication and synchronization of goroutines. The word 'channel is a good description of how a channel operates. Data is passed through one end of a channel, and is received at the other end. Receiving from a channel is blocking; the goroutine will cease activity until a different goroutine passes a value to that same channel. This not only allows communication between goroutines, but also synchronization since there can be multiple listeners on the same channel. Communicating through channels offers a way to guarantee that threads working in parallel are in a known state.

2. The Study

The purpose of this study was to investigate and create a breakdown of the challenges and rewards of working with each language from a new user's standpoint. Hopefully this research will provide a rough understanding of each language for anyone interested in using these languages. I chose Go and Clojure because they are relatively new to the programming world. Before beginning my study, I searched extensively for any kind of research on both Clojure and Go. I could not find much academic research on either language. Creating a unique contribution was also a factor when choosing these two languages.

To give myself enough data to create a satisfactory breakdown of each language, I considered many applications whose implementations would allow me a lot of valuable experience in each language.

I ultimately chose to implement Dijkstra's shortest path algorithm in Go and Clojure. Dijkstra's algorithm is well known and relatively easy to understand, without being too simplistic. By choosing Dijkstra's algorithm, the experience gained with each language was substantial and the code is easy to understand, all while being within the scope of this project.

Dijkstra's algorithm takes in a graph and a source vertex, and returns a structure containing the shortest path to all other vertices. While Dijkstra's algorithm is very serial, it can be run in parallel to solve the all-pairs problem. This does not need any communication between threads. The all-pairs problem is simply solving the shortest path between any two vertices in the graph. In short, Dijkstra's algorithm is run on every vertex and a distance table that contains the shortest path between any two given nodes is created. This process is fairly expensive, as it runs in $O(N^3)$, where N is the number of vertices contained in the graph [10]. In the parallel version Dijkstra's algorithm, the work is split up between *P* tasks or processors. Each process is given *N/P* vertices to analyze. With this division of work, execution time can be reduced to $O(\frac{}{})$.

To implement this algorithm, I had to create a small graphing package for each language. This package had a few simple requirements:

1. Be able to parse a text file containing weighted edges
2. Represent said edges as a graph
3. Keep track of valid vertices

Graphs are usually represented in one of two ways, an adjacency matrix or an adjacency list. For a graph containing *n* vertices, a weighted matrix is somewhat expensive to represent, since it always takes up an *n x n* array of some number format. Adjacency lists are cheaper for sparse graphs; for a graph containing *e* edges, the adjacency list takes up *e* entries of a numeric value and an identifier. I took the less memory intensive approach and represented the graphs by adjacency lists. This meant each vertex had a list that contained a set of adjacent vertices, as well as the costs to get to those vertices. This structure is visually represented in Figure 1.

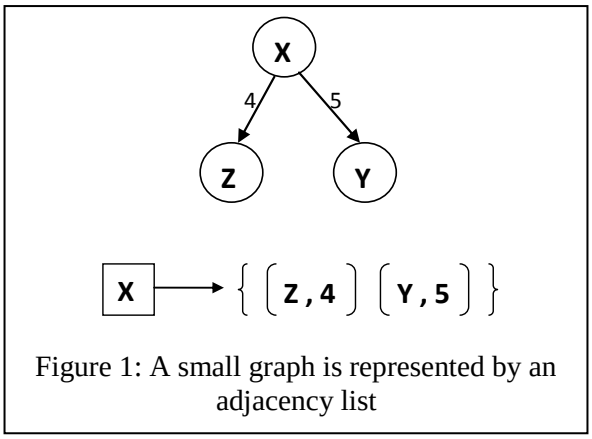

Figure 1: A small graph is represented by an adjacency list

The text files that contained the graph data were a format I made myself for simplicity. Edges in the graph are simply represented by identifying the source node, the end node, and the edge weight, like so:

X Y 5

X Z 4

During the process of implementing Dijkstra's algorithm in each language, I documented my experience of working with each language. I tried very hard to keep each evaluation unbiased, but by the end of the project I definitely had my preferences between the languages. The point of this study isn't to declare one language superior than the other, it is to document the challenges and rewards a new user experiences when first using these languages. This includes all considerations of the programming language, such as its syntax, data structures, available libraries, and available documentation.

4. Implementations

Over the next two sections, I will describe, in detail, the process of implementing the aforementioned applications in each programming language. This section is dedicated to documenting both the interesting attributes of each language, as well as the challenges I encountered. Again, the purpose of this review isn't to declare one language superior to the other, but to document things that set them apart from other programming languages and provide understanding for those unfamiliar with either language.

4.1 Go

4.1.1 File IO

Implementing the graphing package was pretty straightforward in Go. The ioutil package that is provided by the standard io library was very simple to use; it simply parsed the file into a string. I had intended on changing it so that it did a buffered read, but manipulating the string was so easy in Go that I dared not change my already working code.

However, the io package does offer an intuitive and interesting pipe read/write that I am going to take into consideration for future work. The read/write pipe is a good example of how channels that can be made use of. The read and write functions use a channel to sync operations. Once the program has read in a value from the pipe, the read() function signals the channel that is shared by the write() function. The write function wakes up upon receiving the signal, reads in the next portion of the file, and waits for the read function to signal it again. This is a very clever way to incorporate concurrency into an I/O function.

To represent the graph, I first created a "node" structure. The node contained two variables, a string to identify the node, and an integer to store the weight associated with the edge. For directed graphs, only one node has to be built containing the destination node and the weight. For directed graphs I created two nodes, that way each vertex had an identical edge that corresponded to one another. To represent the entire graph I created a slice of linked lists. Each element in the slice represented the adjacency list for a particular node.

4.1.2 Go Slices

Slices are incredibly useful data structures that are not often found in compiled languages. They make it easier to work with arrays and references to arrays. However, since slices are an abstraction for a static data structure, it is necessary to manually manage the length of the slice as well as the size of the array. Slices can point to any segment of an array and be any size as long as it is within the limits of the array. Slices have two important properties, length and cap. The length is the current size of the slice and the cap is the size of the underlying array. One useful tactic is to start the slice size at zero, and adjust the size of the slice every time you need to add a new value to the array. This is an easy way to keep track of how many elements are actually being used in the array, since the built-in len method provides this value. Figure two provides a better look at the relationship between a slice and an array.

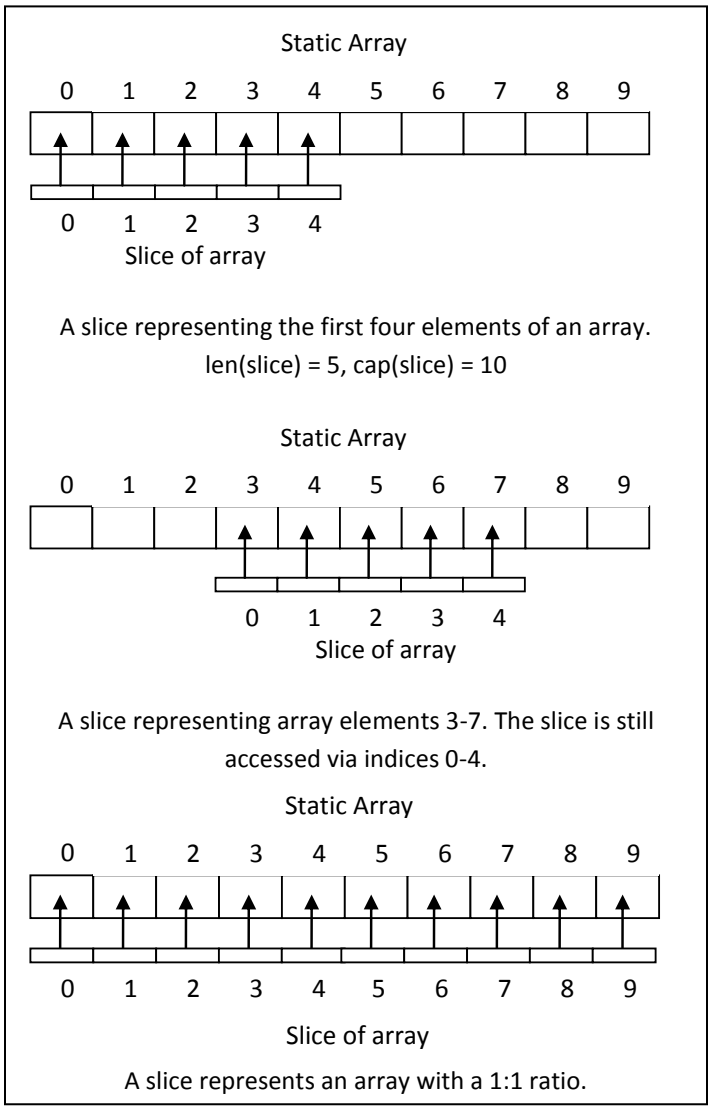

Figure 2

While Go slices are generally easy to work with, they still have the same problems as a static array. Re-allocating a slice/array combination is still just as expensive and requires the programmer to create a function to do it. Since arrays are pass-by-value, the slice is passed to the array. I chose to have the reallocate function make the array twice as large. This exponential approach should ensure that reallocation does not take place too often. After the new allocation is made, the slice values are then copied into the new slice using slice's built-in copy method. This simple but somewhat tedious process is outlined in Figure 3. One interesting thing to note is that the reallocate function I defined must know the type of slice it is being handed. This can be made more generic by having the reallocate function take slice of type 'byte.' This requires casting the structure to a slice of bytes before passing it to the function and casting it back once you receive it back from the function.

```
func reallocateGraph(slice []list.List) []list.List {
	l := cap(slice)
	// Allocate for cap*2, since reallocating too often can be expensive
	newSlice := make([]list.List, l*2)
	// The copy function is predeclared and works for any slice type.
	copy(newSlice, slice)
	return newSlice
}
```

Figure 3: Reallocating a slice in Go Code

4.1.3 Go Maps

The graph could have been stored as a map of adjacency lists, but I wanted to get a feel for how slices operate since they seem to be a staple of programming in Go. Instead, I kept a map that mapped vertex IDs (a string) back to a slice index (an integer). This is a seemingly roundabout way to do things but, it actually worked out to my advantage since Dijkstra's algorithm benefits from having a structure that contains a list of operable nodes.

The map that I used to keep track of operable vertex IDs was named the idMap. The idMap was passed to the Dijkstra function along with the graph. In addition to the idMap, I used a map to keep track of minimum distances between vertices. A 'distance' map was the main data structure that the Dijkstra's algorithm worked to update and return. During a single iteration, the algorithm searches for the lowest distance contained in the distance map given a set of keys provided by the idMap. It then selects the vertex with the smallest distance to work on and removes that vertex from the idMap. Removing the vertex from the idMap causes the vertex to never be looked at again

Working with maps was a familiar experience, up until I realized that maps were reference types. Removing elements from the idMap caused problems whenever the Dijkstra function was called more than once, i.e. in parallel. Unlike slices, maps do not have a built in copy function in Go. This was disappointing, as I had to write a function whose only job was to declare a new map and copy over data. This function is called whenever I call any function that modifies idMap. In figure 3, I call it twice: once when I call 'runDijkstra' and once when calling the 'dijkstra' function. This ensures that each process will get a unique copy that can be modified and thus will not have any side effects.

Of course, one still needs to be careful when passing a reference type to a goroutine, as I found out the hard way. When I carelessly passed the idMap to different goroutines, it took me a long time to figure out why some goroutines were exiting early and with incorrect distance values. Once I solved this problem, I thought of Hickey's "Are We There Yet" keynote on how it can be extremely difficult for programmers to predict the values of references types, since the symbol is often mixed up with what they represent. This is especially difficult when there are many potential processes affecting the value behind a reference elsewhere in the code.

Unlike Hickey's philosophy, I don't share the belief that shared memory is a completely bad thing when used responsibly. Certainly there are many cases when shared memory can be difficult, if not impossible to determine the state of without the use of a debugger. Channels are one way of addressing this issue; having goroutines block on a channel ensures synchronization

between threads. When used properly, they can potentially help the programmer associate the symbolic references to the actual entities.

4.1.4 Channels and Concurrency

To parallelize Dijkstra's algorithm in Go, work is split up into *P* goroutines. Each goroutine receives a partition of size *N/P* vertices to work on. From there, the goroutines run the algorithm on every vertex in its partition. These goroutines utilize channels to communicate the single-source shortest-path distances back to the main thread after every execution of Dijkstra's. Meanwhile, in the main thread, a loop blocks on the shared channel until every vertex's shortest-path distance are communicated. Figure 4 documents the entire process of communicating through channels.

Channels were very straightforward to use. I was greatly satisfied with their simplicity, as well as their intuitive nature. "Channel" is certainly an excellent visual representation of how the channel works; values are passed to one end and come out the other. They were so simple to use that there isn't much more I can say about them other than they are an excellent way to synchronize information between threads.

```
//In the Main Function
   //Create the Channel
   c := make(chan map[string]float64)
   //Loop through list of partitions
   for g := partitions.Front(); g != nil; g = g.Next() {
      //Run Dijkstra's Algorithm for each partition concurrently
      go runDijkstra(g.Value.(map[string]int) ,graph, copyMap(idMap), c)
   }
   for i :=0; i < len(idMap); i++ {
      //Receive a value from the channel
      d := <- c
      for j, _ := range d {
                  //Add d to global distance map
         }
      }
}
func runDijkstra(subset map[string]int , graph []list.List ,
                  idMap map[string]int  , c chan map[string]float64){
      var dist = map[string]float64{}
      //Loop Over each node in the subset
      for i, _ := range subset{
            dist ,_ = dijkstra(graph , i , copyMap(idMap) , nil)
            //Pass the distance map for node i back to the main loop
            c <- dist
      }
      return

}
```

Figure 4: Communicating with channels in Go Code

4.2 Clojure

4.2.1 Maps

To build the graph from a file, the Clojure code uses a library called duck-streams, an abstraction over the java buffered reader/writer. Unlike the Go file reader, this one actually streams the file line by line so that larger graph files can be accommodated. The process-file function takes in a filename, a function to apply to each line, and the hash-map type so that it can initialize a hash map. To represent the graph, I used a map of maps; each unique vertex had its own hash map to store the adjacency list (now adjacency map). Adding onto adjacency maps required the use of merge, which is a function that combines two maps. Creating and accessing the maps is very simple in design.

To keep track of the minimum distances, I used another map just like in the Go implementation. While both programs made use of maps, it is important to note that manipulating the maps were two entirely different experiences. Maps in Clojure are immutable, so assigning any values to a certain key must be done so through an assoc. Assoc does not modify the existing map, but instead returns a new one with the new value merged in. Thus assoc, and all of the other functions I used in Clojure, can be described as pure functions.

4.2.2 Reduce

Modifying maps in Clojure used much different techniques from those used in Go or any procedural programming languages for that matter. To loop through the elements of a map, a ‘reduce’ is what is traditionally used in place of a procedural loop such as ‘for’ or ‘while’. Reduce, also known as a fold, is a common functional programming language idiom used to apply an arbitrary function to a data structure. A reduce is passed three functions, a function, a data set (optional), and a sequence that can be iterated over. Reduces were initially very confusing for me, since I had little prior experience with functional programming. To ameliorate the process of understanding reduces for anyone unfamiliar with them, an example of a simple reduce function is represented visually in figure 3.

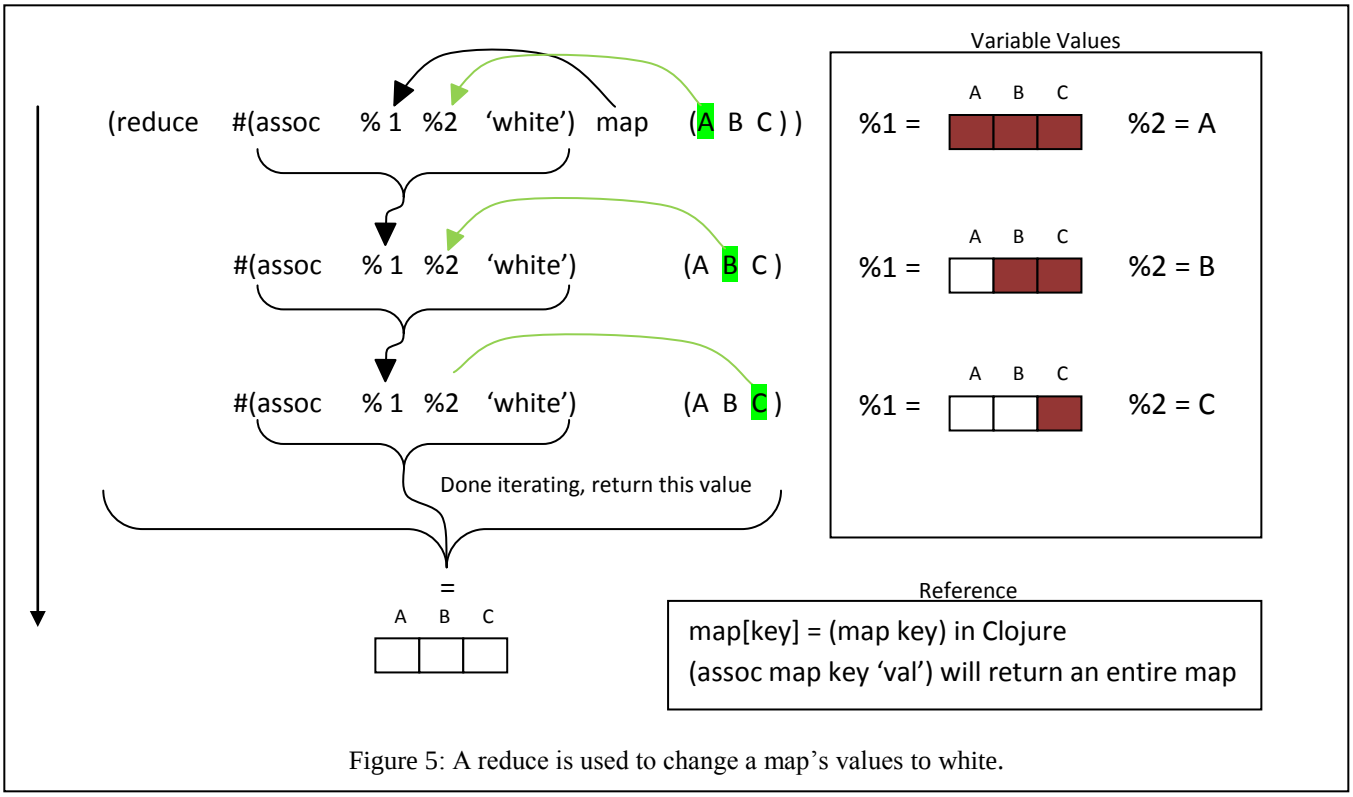

Figure 5: A reduce is used to change a map's values to white.

One particularly elegant function that uses a reduce function is the find-minimum function I created. Clojure provides many utilities for maps that are sorted by key. However, Clojure does not provide functions that do any manipulations with the values held by the keys. There was neither a find-minimum-value, nor a function that found a minimum value and returned the corresponding key. This is most likely due to the fact that maps are not declared to hold one specific value type; it would be impractical to create a function that tried to find a minimum value, regardless of the type of value.

Whatever the case, Dijkstra's algorithm requires a function that will return the vertex with the smallest distance, regardless of whether or not that distance is unique. This gave rise to the 'find-minimum' function featured in Figure 4. Unlike the reduce in Figure 3, this reduce only has a function and a sequence. This just means that during the first iteration, %1 and %2 are assigned the first two values in the sequence. After that, it follows the flow featured in Figure 4. I thought this to be an elegant solution for what would usually take a few variable declarations and temporary values in a procedural language.

```
(defn find-minimum [dist keyseq]
  (reduce #(if( > (dist %1) (dist %2) )
             ;;return %2
             %2
             ;;else return %1
             %1)
   keyseq)
)
```

Figure 6: Code sample of find-minimum

All of the functions I made for this program utilize a reduce function, with the exception of the file parser and Dijkstra's algorithm. Dijkstra's algorithm uses a 'loop' function, while the file parser uses the 'let' function.

4.2.3 Lists and (not) lists

To provide a set of operable nodes for Dijkstra's, I used the 'keys' function to obtain a list of keys from the graph map. This would have been perfect solution for keeping track of which nodes were visited—had the 'remove' function worked on it. Remove is a function that takes a list and a value and returns a list, which omits the specified value. However, the 'key' function returns not a list, but a "Persistent Map Key Seq" (PMKS) data type. PMKS satisfies the list type. However, functions reserved for list manipulation that were passed a PMKS, failed to actually perform any operations on it.

To remedy this, I wrote my own code that 'removes' elements from a PMKS in a reduce statement. This code was awkward and clunky compared to the other functions, but worked correctly. In retrospect, a 'loop' function might have been a more elegant solution, or I could have tried to build a list. It seems as though PMKS's are only good for iterating over. Regardless, I was a bit perturbed by the fact that the PMKS type was incompatible with the functions designed to modify lists.

4.2.4 Parallelizing Dijkstra's Algorithm

Unfortunately, due to time constraints, a parallel implementation of Dijkstra's was not completed. For the purpose of future work, I am including the steps needed to implement parallel version of Dijkstra's algorithm for the purpose of solving the all-pairs problem.

To parallelize Dijkstra's algorithm for Clojure, similar steps must be taken to that of Go's parallelization. First, the vertices N must be divided evenly into sets of N/P, where P is number of desired processes, usually the number of processors. The data structure returned by this process should be a list of key sequences or a list of lists.

An agent that contains the global distance map must then be created. This agent should be modifiable by all threads asynchronously. Once this is done, a unique agent must be created for each subset of vertices. These agents should be in an iterable list. Then, the main loop should iterate though each agent and pass it a function that will:

1. Run Dijkstra's algorithm on each vertex in the subset contained by the agent
2. Update the global agent with the distance data returned by Dijkstra's algorithm by sending it either an assoc or a merge

Meanwhile, the main thread waits for each agent to signal its completion. Once the agents have completed, the main thread can then view the global distance agent. In reality, this process will use $P+2$ threads: P processes to run Dijkstra's, one process that updates the global distance map, and the main thread. This is slightly different from the Go implementation; Go's channels allow for the main thread to block on a channel rather than a thread's completion. Blocking on a thread's completion is circumvented in Clojure by having the global agent.

5. Conclusions

5.1 Go

In summary, I am very impressed with both languages. With Go I was impressed by how intuitive Rob Pike et al. successfully made it. It took only a few days with Go to become comfortable with writing code in it. Perhaps this is because of Go's distinct likeness to C++, which I have considerable experience in. Because I worked in Go first, I found that most of the challenges from working with Go did not come from any of the language's properties or utilities, but from the challenge of understanding how to create my own graphing package that would be easy to use when implementing Dijkstra's algorithm. As I mentioned in the Go section, the only real problem in my implementation came when I was careless and passed around a ref that got modified by different threads, which is a problem not associated with Go.

The type system that Go uses is one of the best I've ever used. I didn't have cause to use a user-defined type, but working with them seems to have numerous benefits. For example, instead of creating subclasses such as those in Java, Go allows you to create a sub-type that can satisfy the functionality of the super type. In the future, I would like to have a more in-depth experience with the OO side of Go.

I would highly encourage anyone interested in Go to give it a try. I am guessing that most users will retreat back to the comfort of their preferred programming language. In fact, that was my first instinct. I kept looking for features I would use in C++, becoming frustrated when Go gave me compiling errors. One example of this was when I tried to manually manage arrays, disregarding the slice type simply because I had never used it before. When it wouldn't let me pass arrays by reference (Side note: Why make '*[]int' a valid type then when you can't do anything with it?) I grumbled and learned how to implement the slice for my array needs. I realize now that using slices is much easier than trying to manually manage arrays. The more I used Go, the more it grew on me. I genuinely enjoyed programming in it.

Of course, there are probably plenty of applications where Go is lagging behind C or C++. For one, it is not nearly as portable. However, Go has the support of Google. According to Rob Pike, quite a few Google teams have picked up using Go as their primary language [12]. With this sort of support, I believe it has the potential to become one of the more popular system languages. In fact the GoLang team made an announcement on December 2nd, 2010, that the Go compiler is going to be part of the 4.6 gcc release [13].

5.2 Clojure

Going into the Clojure implementation, I wasn't sure what to expect. My only experience with Lisp was with emacs lisp. I figured since I understood that, I would have an understanding of Clojure. I was completely wrong. Going from using strictly procedural languages to a Lisp language in the course of a few weeks was one of the most challenging transitions I've had in my programming career. Around three-quarters of my time with Clojure I spent learning about Lisp and the functional approach to solving programming problems. I tried to keep this out of my results as best I could, but I can't help but feel some of my findings on the language are due to the fact that I had no real previous experience with functional programming

languages. Regardless, I hope that my experience with Clojure is insightful for any programmer who, like me, is mostly familiar with procedural languages with a desire to expand horizons.

I would have loved to implement a parallel version of Dijkstra's algorithm in Clojure, but the time it took to learn Clojure was significant. I spent around twice as long on the Clojure code than I did the entire Go code, without actually implementing any concurrency features in Clojure. As I mentioned in the implementation section, I did develop a plan for parallelizing the Dijkstra's algorithm through the use of agents. However, by the time I finished the regular implementation of Dijkstra's algorithm, I did not have enough time left to clumsily fumble with the concurrency features of Clojure.

The huge Lisp learning curve aside, my experience with Clojure has been extremely satisfying. Thinking about programming problems in a functional way and then realizing a solution through Clojure code was a thoroughly rewarding experience. There is certain eloquence and terseness about programming in such a purely functional language such as Clojure. For example, my Go code took around 273 lines, omitting the code it took to implement the concurrency. The same code in Clojure took 78 lines. (On a side note, if lines of code were indicative of the time it took, I think these numbers would be switched around)

6. Related Work and Useful Resources

6.1 Kraus et. al

In "Multi-core Parallelization in Clojure – a Case Study," Johann M. Kraus et al. did a study on Clojure by implementing an algorithm that was specifically designed to benefit from having multiple threads operating on a multi-core processor [8]. They chose Clojure because it is a functional programming language and allows for implicit parallelization. Implicit parallelization is the parallel processing of functions that are free of side effects i.e. Clojure's agents. Explicit parallelization is the process of explicitly defining how threads will communicate and synchronize information, like locks. Kraus et al. utilize Clojure's concurrency features to implement a K-means cluster algorithm. A k-means clustering algorithm takes data and partitions it into *k* clusters based on similarities that the data share.

The benchmarks showed significant improvement when using a parallel multi-core implementation in Clojure, all in 100 lines of code. Kraus et al. commended Clojure for using a software transactional memory, saying that "[t]he design principle of the STM reduces the additional overhead required by the design of parallel algorithms." They also describe the process as requiring "minimal additional effort" in designing software. Conforming to Hickey's beliefs, using pure functions also led to less bugs during the process of implementing the parallel k-means algorithm.

6.2 Wide Finder 2

More excellent research on Clojure can be found on Tim Bray's website, tbray.org. Under a section titled 'Concurrency,' Bray has research notes on all types of programming related to concurrency with a special focus on Clojure. Within the Clojure niche of the research, Bray covers a wide variety of topics ranging from concurrent I/O to 'NOOb tips,' a resource for absolute beginners using Clojure.

One of Bray's larger projects is the Wide Finder 2 project. The original Wide Finder project was to create a concurrent program that reads a large Apache logfile and determines which articles had been fetched the most [14]. Bray's implementation was planned to be in Erlang, but he quickly ran into problems. Luckily, many other programmers caught wind of the project and submitted their own implementations in many different programming languages. Unsatisfied with his own implementation, he later rebooted the project, calling it Wide Finder 2. He chose to write wide finder 2 in Clojure. He found that the performance was average, but did have a lot of interesting findings on Clojure.

Bray's experience with Clojure was very similar to mine. He had a difficult time thinking in a functional manner, but had a great appreciation for the language. In another article titled "Eleven Theses on Clojure," he states that Lisp code is difficult to read for most programmers, himself included [15]. He asserts that most programmers are more familiar with procedural languages, which is true in my experience. However in the same article, Bray calls Clojure the best Lisp ever. He says that other Lisps are "hobbled by lackluster libraries and poor integration with the rest of the IT infrastructure." Running on the JVM, he argues, "makes those problems go away."

6.3 Others

There are not many academic papers regarding Go or Clojure (Go especially), but there is great support for either language on their main websites which are listed in the references section. The websites for both Go and Clojure offer excellent documentation on almost every subject that a programmer might want to know when writing code. The websites really made this study possible. Each language also has a Google group. These groups contain lots of useful information such as sample code and applications created by followers of the language.

7. Appendices

7.1 Go Parallel Performance

Before analyzing the performance of the serial and parallel versions of Dijkstra's algorithm, I made a few optimizations. To begin with, I changed the file IO so that it would do a buffered read instead of parsing the entire file into a string. This was to accommodate for large graphs. Then I removed all instances of maps from the program. The graph was still represented by a slice of linked lists, but I assigned vertices a number so that each vertex would have a unique index in the slice. A bit map replaced the ID map, and the distance vector for each vertex was also replaced by a slice of type integer. Removing all the maps from the program removed all the performance flaws associated with accessing a map, i.e. searching through keys, copying maps, etc.

This gave me a total of four versions: my original serial and parallel implementations of Dijkstra's algorithm, and the new serial and parallel implementations. To test the performance of each program, I ran them on graphs that I generated. The graph generator I used I also built in Go. I generated 6 graphs of varying size, ranging from 250 to 12,000 vertices. Each program received the same copy of each graph.

To measure performance, I used the UNIX utility 'time' to measure the time each program took. These tests were performed on a CSEL computer which has an Intel Xeon Dual-Core CPU at 2.66ghz. Due to time constraints, I choose not to implement the old code further than 500 vertices. Their inclusion is mainly to demonstrate the superiority in performance of the new code. The results can be seen in the table below.

Table 1: The performances of each implementation of Dijkstra's algorithm on graphs of varying size

		250 vertices/ 500 edges (10 samples)	500 vertices/ 1000 edges (10 samples)	1500 vertices/ 3000 edges (5 Samples)	5000 / 7500 (2 samples)	10,000/ 12,500 (1 sample)	12,000 / 15,000 (1 sample)
Parallel	real	0.3264s	1.336s	23.480s	9m11.461s	23m45.653s	73m2.208s
	user	0.3168s	1.336s	23.457s	9m11.302s	23m45.373s	73m0.254s
	sys	0.0052s	0.0032s	0.0100s	0.006s	0.016s	0.024s
Serial	real	0.3243s	1.444s	23.845s	9m17.981s	24m12.631s	74m22.668s
	user	0.3200s	1.462s	23.831s	9m17.869s	24m12.255s	74m21.815s
	sys	0.0048s	0.0028s	0.0136	0.008s	0.012s	0.008s
Parallel (old)	real	2.328s	14.566s				
	user	2.319s	14.517s				
	sys	0.0056s	0.0028s				
Serial (old)	real	2.151s	14.155s				
	user	2.137s	14.145s				
	sys	0.0048s	0.0040s				

In all cases (except for the old programs), the parallel implementation of Dijkstra's algorithm to solve the all-pairs problem outperformed the serial implementation. The differences in the smaller graphs could have been due to fluctuations in processor activity, but performance on the larger graphs is slightly better when using the parallel algorithm. In all cases, the parallel algorithm used less CPU time than the serial algorithm. The percentage decrease in CPU time from the serial performance is documented in Table 2.

These results were surprising, since it is often the case that serial code outperforms parallel code due to the costs of communicating data between threads. Indeed the thesis committee hypothesized that gains using this implementation would be unlikely. However, it appears as if the Go concurrency model is quite effective in reducing the cost of this communication.

Table 2

Graph Size	CPU time decrease using parallel algorithm
1500	1.53%
5000	1.18%
10,000	1.85%
12,000	1.83%

An interesting point to notice is that the serial version of the old implementation significantly outperformed the parallel implementation, despite the use of goroutines and channels being identical. I believe this is due to the fact that the channels in the old code were being passed very large maps, which could create a lot more overhead than passing arrays of integers. In short, maps slowed down not only the serial portions of the code, but also could have hindered any potential performance gains provided by Go's concurrency features simply because of their size.

Sources Cited

[1] Goth, G., "Entering a parallel universe," *Communications of the ACM,* September 2009, pp. 12-17. [Online] Available: http://portal.acm.org/citation.cfm?id=1562164.1562171&coll=portal&dl=ACM

[2] Sun, X. and Chen, Y., "Reevaluating Amdahl's law in the multicore era," *Journal of Parallel and Distributed Computing,* February 2010, pp. 183-188. [Online]. Available: http://www.sciencedirect.com/science?_ob=ArticleURL&_udi=B6WKJ-4WBT454-1&_user=918210&_coverDate=02/28/2010&_alid=1243790077&_rdoc=1&_fmt=high&_orig=search&_cdi=6908&_docanchor=&view=c&_ct=21&_acct=C000047944&_version=1&_urlVersion=0&_userid=918210&md5=f4a1a1603beb57ddc27bc2e7e843b6ea

[3] "Lesson: Concurrency." *The Java Tutorials.* Web. Accessed: March 10, 2010 http://java.sun.com/docs/books/tutorial/essential/concurrency/

[4] Hickey, Rich. "Rationale." *Clojure. Web.* Accessed: December 2, 2010 http://clojure.org/rationale

[5] Nikishkov, G. P., Nikishkov, Yu. G., and Savchenko, V. V., "Comparison of C and Java performance in finite element computations,"*Computers & Structures,* September 2003, pp 2401-2408. [Online]. Availiable: http://www.sciencedirect.com/science?_ob=ArticleURL&_udi=B6V28-49FR4BN-1&_user=918210&_coverDate=09/30/2003&_rdoc=1&_fmt=high&_orig=search&_sort=d&_docanchor=&view=c&_acct=C000047944&_version=1&_urlVersion=0&_userid=918210&md5=c6fa9394e2491fe967ebfd5d7da31614

[6] Pike, Rob. "The Go Programming Language." October 30, 2009 Google Tech Talks. [Online]. Available: http://www.youtube.com/watch?v=rKnDgT73v8s

[7] "Installing Go" *The Go Programming Language.* Web. Accessed: December 2, 2010. http://golang.org/doc/install.html

[8] Kestler, H. A., Kraus, J. M., "Multi-core parallelization in Clojure: a case study," *European Conference on Object-Oriented Programming,* 2009, pp 8-17. [Online]. Available: http://portal.acm.org/citation.cfm?id=1562870

[9] Hickey, Rich. "Are We There Yet?" February 25, 2010. Keynote Address. [Online]. Available: http://groups.google.com/group/clojure/files

[10] Foster, Ian. "Case Study: Shortest-Path Algorithms." 2005. Web. Accessed: December 7, 2010.
Available: http://www.mcs.anl.gov/~itf/dbpp/text/node35.html

[11] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. "Lonestar: A Suite of Parallel Irregular Programs." Proceedings of the *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. pp. 65-76. April 2009. [Online] http://iss.ices.utexas.edu/Publications/Papers/ispass2009.pdf

[12] Pike, Rob. "Another Go at Language Design" July 22, 2010. Keynote Address. [Online]. Available: http://assets.en.oreilly.com/1/event/45/Another%20Go%20at%20Language%20Design%20Presentation.pdf

[13]Taylor, L. I., "Gccgo now committed to gcc mainline" December 2, 2010. Online Announcement.
Available http://groups.google.com/group/golang-nuts/browse_thread/thread/200979c143e959fc

[14]Bray, Tim. "Wide Finder 2," December 9, 2009. Web. Accessed: December 9, 2010. Available: http://www.tbray.org/ongoing/When/200x/2007/09/20/Wide-Finder

[15]Bray, Tim. "Eleven Clojure Theses," December 4, 2009. Web. Accessed: December 9, 2010. Available: http://www.tbray.org/ongoing/When/200x/2009/12/01/Clojure-Theses

More information on Go and Clojure can be found on their main websites www.golang.org and www.clojure.org respectively. For latest information from the Go and Clojure communities, visit their respective Google groups at http://groups.google.com/group/golang-nuts and http://groups.google.com/group/clojure. Also check out www.reddit.com/r/clojure and www.reddit.com/r/golang for the newest articles on Clojure and Go.