Multi-Threaded Operating System

Final Report

Submitted for the BSc in Computer Science

April 2016

by

Alexander Matthew Robinson

Word Count: 16223

Table of Contents

1 Introduction

In recent years we have seen a trend with microchip manufacturers wherein they have moved from single core microchips with high clock rates, to so called 'massively multi core' architectures. This shift has occurred in an effort to continually increase performance gains from CPU development as the main technique previously used, increasing the transistor count and clock frequencies of CPU's, has begun to see a decrease in effectiveness due to CPU's reaching the so called “power wall” (University of Illinois, 2008).

Although this shift has enabled processors to continue to follow the trend laid out within Moore's Law, which states that transistor density within CPU's will double every two years (Moore, 1965), it has also meant that software engineers have needed to modify the way in which they approach software development for systems such as these. As these systems become more prevalent within today's society, it is therefore becoming more pressing for developers to have the tools to enable them to fully utilise the processing power within CPU's.

As such the premise of the project that will be described in the report that follows, is related to the development of a framework that will enable software developers to capitalise on the processing power available to them through the use of parallelism within their projects.

2 Aim and Objectives

The overall aim of this project was to design and implement a framework that enables users to implement task based parallelism in an efficient and easy to use manner.

In order to complete the project aim stated, the project needed to fulfil the following objectives:-

Objective 1 – Develop a classification mechanism that enables users to define areas of their software as Tasks.

Task based parallelism, is a type of parallelism where the software is separated into distinct tasks such that they can be run independently. As such, one of the key components of this project was to provide users with the ability to classify specific areas of their software as tasks which can subsequently be executed independently.

Objective 2 – Implement a scheduling algorithm to allow multiple tasks to run simultaneously.

One of the other key components of task based parallelism, is the scheduling algorithm which assigns tasks to specific threads. Therefore for the project to be successful it was necessary to plan, design and implement a scheduling system which can efficiently execute varying numbers of tasks simultaneously.

Objective 3 – Design and implement a framework that can be easily implemented within a software project by a user.

To ensure that the end users are able to make the most of the implemented framework, it was necessary to design the classes and other objects of the system in such a way that they can be easily integrated into the end users own software.

Objective 4 – Optimise the framework so that it can maximise its utilisation of the CPU's processing capability.

Ideally the implemented system would utilise 100% of the CPU’s processing capability, however this is not feasible due to the fact that the operating system, as well as other applications, will be utilising the processor simultaneously. As such, for this objective to be fulfilled it will be necessary for the system to utilise a significant proportion of the processor time across all of the available CPU cores when executing.

The following objective is a secondary objective, which will be completed only if the project is not constrained by factors such as time:-

Objective 5 – Produce a framework that implements more than one scheduling algorithm.

It would be useful from a scientific point of view to be able to implement multiple scheduling algorithms within the framework. This is so that it would be possible to directly compare multiple scheduling algorithms within the same code base.

3 Background

3.1 Problem Context

The main focus of this project is the complexity associated with writing software that can run in a parallel environment and how this can be made easier for software developers to implement. Therefore in order to understand how parallelism can be simplified, it is necessary for us to first outline specifically what parallelism is.

Parallelism as defined in the Oxford Dictionary is the “use of parallel processing” (Oxford Dictionaries. Oxford University Press, b). Parallel processing, is a type of computation in which multiple operations are performed in parallel across multiple processors.

It is also necessary to mention concurrency due to how prevalent the word is within the associated literature. Concurrency can be defined as “existing, happening or done at the same time” (Oxford Dictionaries, Oxford University Press, a). Which when applied to the field of Computer Science can be seen to infer that concurrency is the simultaneous execution of computational operations. As can be seen, this is similar to parallelism in that it is related to the simultaneous execution of computations yet is distinct due to the fact that concurrency is a more generalised term. Evidently concurrency and parallelism could be used interchangeably, however for the duration of this report parallelism will be used exclusively.

Parallelism can be implemented in a variety of different ways, however the two ways that most concern this project are Data and Task based parallelism. The former of the two, Data based parallelism, is an approach wherein “parallelism comes from simultaneous operations across large sets of data” (Hillis & Steele Jr, 1986) which can be taken to mean that the same operation is performed simultaneously on a number of different pieces of data. The second of these two styles of parallelism, Task based parallelism, was detailed by Gross, O'Hallaron and Subholk (1994, p. 16) as allowing “the programmer to explicitly partition resources (including processor nodes) among the application modules”. What this statement means, is that Task based parallelism is focused on the distribution of sections of a given application, in the form of “tasks”, to the available resources.

In order to be able to implement any form of parallelism, it is necessary for us to make use of multithreading technology. Multithreading, is a means through which multiple sequences of programmed instructions can be scheduled to execute simultaneously. This is done by encapsulating the instruction sequences within a software thread which is then executed upon the available hardware. Within multi-core architectures, systems are able to utilise multi-threading to capitalise on the multiple processors that are available to them by having several threads executing simultaneously where every processor is executing at least one thread instance. It is worth noting here that multi-threading can also be utilised within a single processor/core environment, wherein time slicing is used to allow threads to execute seemingly in parallel. This however is not parallelism, as the operations do not occur simultaneously they are only perceived as such by the user.

The creation and destruction of software threads is a computationally expensive process. Owing to the fact that threads are automatically destroyed once they complete the execution of their instructions, an application that employs a large number of threads could be relatively inefficient. In order to overcome this issue a process called thread pooling was devised.

Thread pooling is a process wherein a number of threads are created when an application is initially executed, these threads are then utilised for the duration of the applications life to execute the necessary computations. Once all of the necessary computations have been completed, the threads will be destroyed en masse. This method of utilising threads is

beneficial for two reasons, firstly as it negates the aforementioned overhead associated with the creation and deletion of threads. Secondly, as this method has an upper limit on the number of threads available at one time it ensures that the applications utilising them do not incur a loss of performance that is associated with the number of independent tasks exceeding the available hardware resources. In a system that does not utilise the thread pool system, when a new computation needs executing a new thread would be created. If the number of software threads currently executing exceeds the number that can be handled directly by the hardware, the operating system will intervene to ensure that all of the threads have appropriate usage of the resources. This will often come in the form of a time slicing system that limits the time in which a single thread has access to hardware resources, which itself creates an overhead owing to the need to save states and other such tasks when switching between threads (Robinson, 2007).

3.2 Comparison of Technologies

3.2.1 Programming Language Comparison

This project could have been developed using one of several different languages, however for this report only three languages have been detailed within Table 1. These languages have been chosen from the many that are available based upon the experience the developer has in said languages.

Language	Development Speed	Developer's Competence	Most Recent Use	Built In Threading Ability	Performance
C++	Moderate/Low	Moderate	2015	Yes (C++11 onwards)	High
C#	High	High	2014	Yes	Medium
C	Low	Low	2014	No	High

Table 1: Table of considered languages

The first language for consideration is C++. As a language C++ is one of the most popular and widely used programming languages in the world at present (IEEE Spectrum, 2015).There are many good reasons for its popularity including its highly efficient standard library, its ability to work cross platform and the low level nature of some of its features. With that being said there are some drawbacks related to development within C++, most notable is that development speed can be severely hindered by the languages type system (statically typed) and lack of garbage collection functionality. This reduction in development speed can also be compounded by a developer's lack of knowledge within the language. Threading within C++ was only included in the standard library relatively recently, specifically within the C++11 variant of the language. Prior to this it was necessary to include a third party library such as Boost (Boost Software, 2015) which would provide threading functionality.

The second programming language that can be considered is C#. The language itself is a multi-paradigm strongly typed language that was originally developed by Microsoft around the year 2000. Although C# is technically cross platform, through projects such as DotGNU (DotGNU Project, 2012) which provides an open source implementation of C#, the language is primarily used for development within a Microsoft Windows environment owing to the fact that the runtime environment used for Windows machines, known as the Common Language Runtime (Microsoft, 2015 a), is encapsulated within the .Net framework which is closed source and therefore cannot be implemented within other environments. The .Net framework, although closed source, provides a large number of classes through its Framework Class Library one of which is a Threading class. Although C# previously had a reputation for having relatively poor performance, owing to the JIT (Just In Time) compilation mechanism employed within the language, in recent years this process has become more efficient to the point where C# performance can under certain circumstances be comparable to that of C++.

The third and final language that can be considered is C. The language itself is the forebear of the two languages previously mentioned in this section, therefore there are many similarities between them especially with regards to syntax. However the C programming language itself is relatively primitive, owing to the fact that it was originally developed in the late 1960's to the early 1970's, and as such it does not have a built in abstraction of the threading mechanisms provided by the Operating System. Therefore to implement multithreading in the C language it is necessary to use Posix threads, or Pthreads, which is a language agnostic execution model as opposed to a native threading library associated with the language itself.
It was initially envisioned that the system would be implemented in C++, owing to its superior performance when compared to C#. The C language could be utilised for the same reasoning,

but due to the developers limited exposure to the language this decision would be ill advised. However owing to the scale of the project proposed, using C++ could impact upon the success of the project due to the developer's competence which could lead to software being produced at less than optimum speed. As such, the project was implemented in the C# programming language which provided the developer with the necessary tools to implement the framework in a timely manner with only a minor trade off in terms of performance.

3.2.2 Threading Mechanisms

As mentioned previously one of the libraries within the .Net framework is the Threading library. Within this library there are a variety of different means through which multithreading can be implemented within a piece of software. Owing to this, it will be necessary to review these methods to find the most appropriate for this project.

The first method that we must discuss is the Thread class. This is the .Net framework's purest implementation of software threading and as such it functions by accepting a delegate function as a parameter. A new instance of the Thread class is subsequently created and the delegate function is executed upon the available hardware. Upon completion of the execution cycle the Thread instance is destroyed. Evidently this would be an issue given the desire to produce a thread pooling mechanism, however as the Thread class provides some useful methods, such as Sleep and Interrupt that enable us to stop and restart the Thread instance at a given time (Microsoft, 2016 d), the class could be utilised successfully within this application.

Although the Thread class would be the most obvious choice for implementing the proposed system, as it is simply a wrapper of the operating systems own threading mechanism its use introduces issues such as deadlocks and race conditions. A deadlock within this context, would be where a Thread instance is unable to access a shared resource because it is being held by another Thread instance that is executing. Race conditions contrast this, as they are related to the sequence in which operations modify state. Within this project race conditions could occur if access to shared memory, such as with variables or I/O, is not handled correctly. Evidently, if the Thread class was to be used it would be necessary to utilise mechanisms that would enable us to ensure the integrity of any shared memory attributes.

The second method available to implement multithreading into a .Net application, is the pre-built Task parallelism class which utilises an existing ThreadPool class. The implementation of Task based parallelism provided within the .Net framework allows the user to specify a region of code as a Task which can potentially return a value, and then have it execute on a Thread within the associated ThreadPool. Evidently utilising these classes within the application would mean a substantially smaller amount of work would be required for the implementation of this project, however as one of the aims of this project is to produce a system that implements Task based parallelism it would be unethical to utilise software written by someone else to achieve this aim.

The third method available to us through the .Net framework, is the BackgroundWorker class. The BackgroundWorker thread is a means through which developers can "run time-consuming operations on a background thread" (Microsoft, 2016 b). What is meant by this, is that the BackgroundWorker class provides the developer with a system through which they can pass functions that handle processes such as downloads or database transactions which will execute on a Thread that is separate from the main Thread on which the application is executing, seemingly in the background. This method of implementing multithreading, although useful in some contexts, would be inappropriate given the premise of this project.
Finally the .Net framework provides us with the ability to write software that executes asynchronously, wherein a function or set of functions execute in parallel to the main system thread that is executing the application. The async keyword, is a modifier that allows a

software developer to specify that a method can execute asynchronously of the rest of the application. The await keyword, is an operator that can be used within an async method to specify that the operation it is associated with is suspended until the awaited task has completed.

The asynchronous programming method provided by the .Net framework is not dissimilar to the previously mentioned BackgroundWorker, in that it is best suited for systems wherein there is a potential bottleneck stemming from a process that requires a varied amount of time to execute, such as downloading files from servers. As the multithreading part of this is handled by the compiler, this method of execution would be inappropriate for the task at hand as it would limit our access to the individual threads such that we would be unable to deploy them in a thread pool style.

As can be seen, despite being flawed in some respects the most appropriate mechanism that can be used to implement multithreading within this project is the Thread class. This is owing to the fact that it provided the most flexibility for implementing the Task based parallelism in the manner that is required.

3.2.3 Technologies to Implement Mutual Exclusion

As the Thread class was utilised for this project, it was subsequently necessary to utilise mechanisms that ensure mutual exclusion.

Mutual Exclusion as described by William Stallings, is “The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those resources” (Stallings, 2011). This definition can be put into layman's terms as being a requirement of a system that no two processes should access the same shared resources simultaneously. The reasoning for this is to ensure that race conditions do not occur, which is where multiple processes access the same shared data item and the resulting values produced by each process can vary depending on the timing of execution of the processes. The principle of mutual exclusion was initially identified by Edsger Dijkstra in his paper *Solution of a problem in concurrent programming control* (1965).

A means of implementing mutual exclusion within software is to use locks, which are mechanisms that are built into programming languages that enforce limits on access to resources. Generally speaking locks function in a way that it is necessary for the thread to acquire the lock prior to accessing the data associated with the lock, which is known as an *advisory lock*. The simplest type of lock is a binary semaphore, which is a variable that is toggled to detail if the data is currently being accessed or not. This is by no means the only type of locking available, although it is one of the most prevalent.

One of the other locking strategies available is to wait for the resource to become available by making the thread 'spin', which is known as a *spinlock*. The *spinlock* strategy contrasts that of other locks in that this style does not require the requesting thread to return from the task and subsequently request it again. This can be more efficient in scenarios where the locking mechanism is only blocking the data for a short period of time, as it negates the overhead associated with returning to a calling function. However the flip side of this, is that if the thread is 'spinning' for an extended period of time the efficiency of the *spinlock* is lost as the thread cannot perform any other tasks.

Within C# there are four classifications of Locks available to the programmer: Exclusive Locks, Monitors, Mutexs and SpinLocks.

The first of these, is the most primitive of the four and functions by blocking access to the internal data entirely (i.e. without spinning). This type of Lock makes use of functionality from the Monitor class, specifically using the Enter and Exit functions, which means that although the Exclusive Lock is built on top of a Monitor a lot of the functionality of the Monitor is lost and therefore this Lock is more primitive.

The Monitor class is a built in class within the .Net framework and provides functionality such as pulsing, where a thread waiting for an object is notified of a change in state, and waiting, where the thread that holds the lock releases said lock and then blocks the thread from executing anything else until the lock has been reacquired. As can be seen the Monitor classes additional functionality can provide a much more robust multithreading system when compared to the Exclusive Lock, however owing to its more complex nature it can lead to more deadlock style scenarios where multiple threads are waiting on each other to complete.

A mutex is slightly different to the other locking mechanisms available within the .Net framework as it is neither a class nor a wrapper of a class within the .Net framework, rather it is a .Net wrapper of an operating system wide mutual exclusion construct. This therefore means that unlike the other three locking mechanisms, a mutex is not bound to the application domain that it is within (Microsoft, 2015 c). This approach is therefore appropriate for systems that have multiple applications accessing and modifying the same data set.

As was previously mentioned, a SpinLock enables a calling thread to wait for the Lock to be released instead of returning to the calling function. This locking mechanism has increased efficiency when compared to a standard lock, owing to it not having overhead from returning to calling function, however if a C# SpinLock is spinning for longer than “a few tens of cycles, … [the SpinLock] will use more CPU cycles and thus can degrade the performance of other threads or processes” (Microsoft, 2015 d).

Given that the system that has been implemented was contained within one application domain the mutex class was an excessive locking mechanism, although the mutex can be used within a purely local setting this would negate any preference over the native (Exclusive) locks. As well as this given that the Monitor class can result in deadlock style scenarios if not implemented correctly, using this style of locks within the framework could have potentially been ill advised. Finally as some of the sections of the application can require a varying number of processor cycles to execute, using a SpinLock for this activity could negate any performance gained from not returning by waiting in a loop for too many cycles. As such, the main locking mechanism that was used within this framework was the standard Exclusive Lock, although there may have been some scope for the SpinLock to be implemented.

3.3 Comparison of Algorithms

Scheduling multithreaded computations is a complicated task, owing to the need to ensure that the available threads remain busy whilst simultaneously ensuring that the number of active threads does not exceed the amount that can be handled by the available resources. There are a variety algorithms and mechanisms available to aid in the scheduling of computations, however for this project we will employ mechanisms that are classified as dynamically multithreaded. This style of mechanism, is distinct from so called static multithreading as the system contains a scheduler which load balances the computations. This therefore allows programmers to “specify parallelism in applications without worrying about communication protocols, load balancing, and other vagaries of static-thread programming” (Thomas H. Cormen, 2009). The two main scheduling mechanisms within this classification of multithreading are work sharing and work stealing.

Work sharing, is the process of assigning any available work to underutilised processors, upon request, through a centralised scheduling system which makes use of an internal data store

of work. This was detailed in the 1999 paper *Scheduling multithreaded computations by work stealing* (Robert D. Blumofe, 1999), in which it states “In work sharing, whenever a processor generates new threads, the scheduler attempts to migrate some of them to other processors in hopes of distributing the work to underutilized processors”.

The benefits of work sharing, is that firstly it is relatively simple to implement as it is comprised of centralised threads and a scheduler. This however creates issues with regards to scalability, owing to the inability of the system to cope with multiple processors running out of work simultaneously as it will inspect and distribute tasks on a first come first serve basis.

Work stealing contrasts this approach as it places the responsibility of obtaining new work in the hands of the processor instead of being reliant upon the scheduler providing work. This is done through a work stealing mechanism, which requests to “steal” work from other processors, as opposed to requests to the central scheduler. This mechanism is aided by the presence of a local data store within the processor instances, in the form of a deque, thereby ensuring that the processors can operate independently.

Work stealing by its very nature is highly distributed and independent, which therefore means that it scales relatively easily. However, unlike work sharing it is not a trivial system to implement and test due to the presence of the stealing mechanism.

Although work stealing is theoretically superior, in order for this project to fulfil objective 5 it was necessary to implement both algorithms within the final system.

In Robert D. Blumofe's 1995 thesis entitled *Executing Multithreaded Programs Efficiently* (Blumofe, 1995) he details a randomized work stealing algorithm which forms the basis of the work stealing mechanism that has been implemented. However, as the algorithm that is specified in this paper is for the Cilk programming language it was necessary to modify it such that it was appropriate for the scope of the project.

3.3.1 Deque Data Structure

As mentioned previously, the Work Stealing algorithm utilises a deque data structure as the internal data store for the worker threads. A deque is a double ended queue which means that it is an implementation of the queue data structure that allows elements to be added or removed from both the top and the bottom. As C# does not implement the deque data structure, it was necessary to create one for use within this project. There are two common methods of implementing this data structure: using a dynamic array or using a doubly linked list.

A dynamic array, as its name infers, is an array which can modify its size to suit the number of elements within it. It is necessary to specify here that a dynamic array differs from a dynamically allocated array in the sense that a dynamically allocated array has a fixed size when the array has been created, whereas a dynamic array has, at least in theory, no fixed size even after creation of the array.

The resizing procedure utilised within a dynamic array will often be implemented in such a way that the array will resize by a large amount and the additional space will store future additions. This is done so that with each new addition to the end of the dynamic array it is not necessary to resize the array, which could incur significant overhead. Evidently this method is a much more efficient means of expansion, and allows for a dynamic array to add elements to the end of the array with an amortized time complexity of O(1), as opposed to O(N) if it resized the array for each new element.

Although insertion at the end of dynamic array has a constant time complexity, the same cannot be said for insertion at the beginning of the array. This is because when inserting at the beginning of the array it is necessary to move the current n elements of the array. This process therefore has a linear time complexity of O(N). This issue is addressed within deques by implementing a variant of dynamic arrays known as array deques, which allow the array to grow from both ends therefore meaning that the other elements within the array need not be moved. This therefore means that the time complexity of this process is once again has a value of O(1).

As with all arrays, the access time of a specific element within a dynamic array has a time complexity of O(1). This is because it is possible with an array to directly access the element regardless of the size of said element, as long as the position within the array is known.

As has been mentioned, a deque implementation generally utilises a variant of the dynamic array called an array deque. This variant allows the array itself to expand in both directions. There are three common ways in which this can be implemented.

The first, is to implement a circular buffer which is only resized when the buffer becomes full thereby reducing the frequency of the resizing. What this means is that the buffer itself is of a fixed size and is implemented in such a way that the buffer “wraps around”. This specific implementation would require references to the start and end elements to be held such that the “wrap around” functionality does not cause confusion when reading from the buffer.

Another means of implementing this is to begin allocation of space within the underlying array from the centre and subsequently expand the array when either edge is reached. This however does result in more frequent resizing and could also have a larger amount of wasted space when compared to the circular buffer implementation.

The final means of implementing an array deque, is to store the elements in a sequence of smaller arrays which are related to each other through a dynamic array holding references to the smaller arrays. This however has a much larger memory footprint when compared to the previous implementations, owing to the need to create multiple arrays which each have overhead related to them.

A doubly linked list is a variation of a linked list that within each node there are two links, one to the previous and one to the next node within the sequence. This structure handles the start and end point of the data structure by setting the links within these nodes to a terminator value such as null. With this in mind, each node within a doubly linked list contains 3 elements: the link to the previous node, the value to be stored and the link to the next node.

The time complexity associated with inserting into a doubly linked list can be represented in two forms. The first, is O(1) which is representative of inserting into a known location within the list such as at the start or the end. It is a constant time as the location is already known and thus the data structure does not need to be traversed. As such if we were to insert into an unknown point, this would have a complexity of (search time + O(1)) owing to the need to traverse the structure before inserting. However, as this structure would implemented within a deque where only the head and tail can be modified the insertion characteristic would always have a time complexity of O(1).

There are two implementations of a doubly linked list that can be used to implement a doubly linked list, namely open and circular. The former, is where the head and tail nodes of the list point to terminators for the previous and next nodes respectively. The latter contrasts this, by having the head and tail node link to each other in a circular like structure, hence the name. For the deque structure, an open implementation is more appropriate as the circular referencing used in the other system is redundant in this scenario.

Owing to the fact that the performance characteristics between the two data structures are almost identical, it was necessary to choose between these two data structures based upon the ease of implementation. As such, the most appropriate means to implement the deque data structure was through the use of an open doubly linked list.

4 Designs

4.1 System Designs

The project is comprised of three individual components that when combined produce a cohesive system. The components are the scheduler, the worker threads and the task; the way in which these components integrate can be seen in the Figure 1 below.

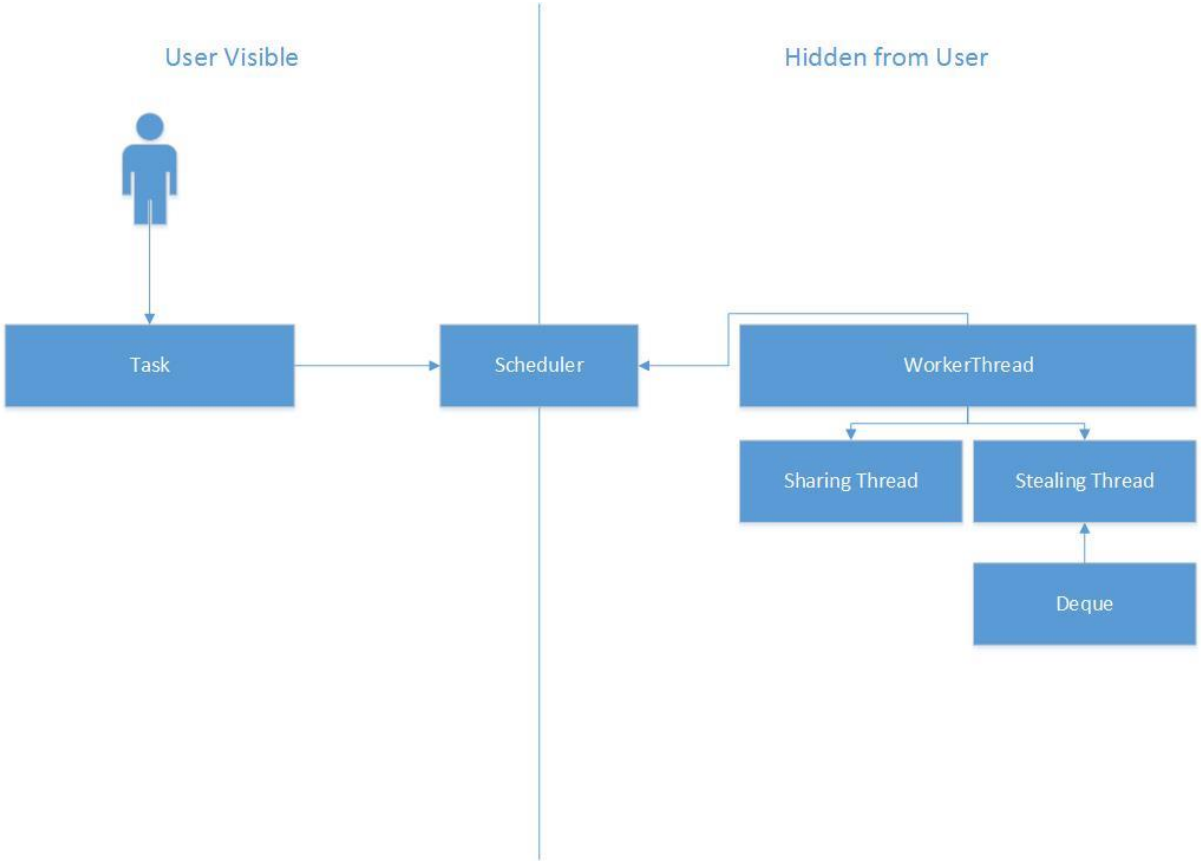


Figure 1: System Architecture

As can be seen the user only has access to the Task and Scheduler objects, although the users access to the latter object is very limited. This was an intentional design choice as allowing the user access to the inner workings of both the Scheduler and Worker Threads could lead to the system functioning at a less than optimal way.

As well as this, the Worker Thread object can be seen as an abstract class as opposed to a true class. This is due to the need for the system to implement two differing scheduling mechanisms, which therefore require two distinct classes which share a common backbone.

4.1.1 Component Design

4.1.1.1 Scheduler

The Scheduler class within this project, will implement the Singleton design pattern (Gamma, 1995) such that only one instance of the Scheduler class will exist within any piece of software that implements this system. The reason for this is that if a number of Scheduler instances existed, there is the potential for the number of Threads executing on a single processor to exceed the limit that can be handled by the processor which could conflict with the objective to efficiently use the CPU's full capability.

The Scheduler itself will be composed of three sections; general, work sharing and work stealing. This is due to the fact that although there are two Scheduling algorithms being implemented, there will only be one Scheduler class. As such there are sections for the individual scheduling algorithms as well as a section for the methods that the two other sections are reliant upon (such as for starting and closing the system).

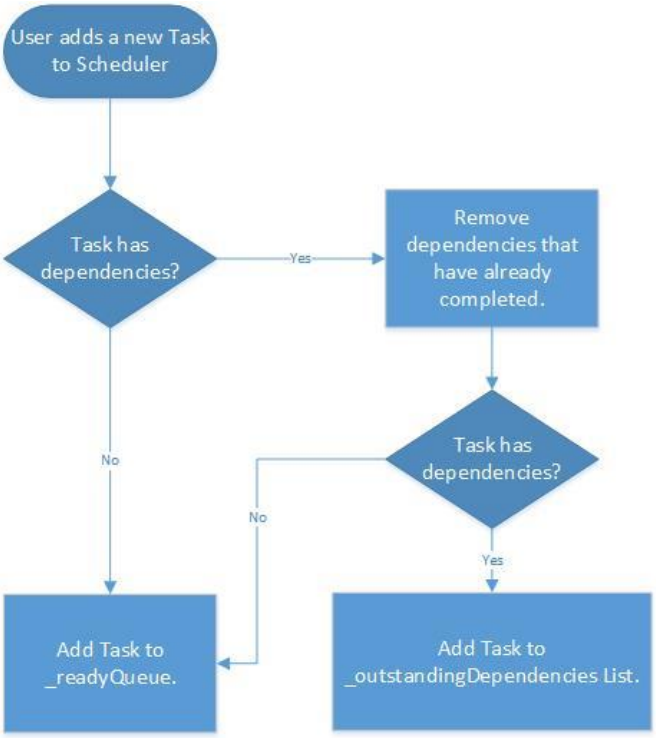


Figure 2 - Adding a Task to Work Sharing Scheduler

The diagram above represents the process that takes place when a new Task object is added to a scheduler that is using the work sharing algorithm. As can be seen, the first process that takes place is that the scheduler requests the number of dependencies that the Task object has. If this value is 0 then the Task is added to the *_readyQueue*, which is the scheduler's internal data store for Tasks that are ready to be executed. Elsewise the scheduler will remove any dependencies from the Task that have already completed, thereby ensuring the Task is up to date. From here the number of dependencies is once again requested, and if the this

value is still greater than 0 then the Task object will be added to the *_outstandingDependency* list to await the completion of its other dependencies.

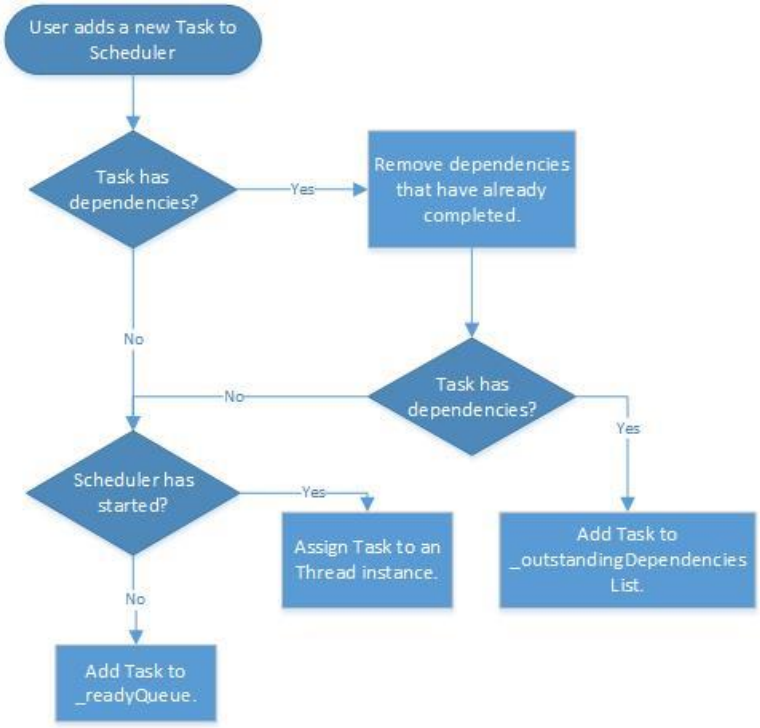


Figure 3 - Adding a Task to a Work Stealing Scheduler

The approach taken when adding Tasks to the scheduler under the work stealing is similar in many respects to the work sharing approach, but differs in the way it handles Tasks after they have been stripped of any dependencies. As can be seen from the above diagram, the work stealing algorithm has an option as to how it should proceed depending upon the current state of the scheduler. If the scheduler is executing, then the Task will be directly assigned to a worker thread that is chosen at random. If, however, the scheduler is not currently executing then the scheduler will add the Task to *_readyQueue* for temporary storage until the scheduler is started.

Figure 4, below, represents how the scheduler will assign Tasks under the Work Sharing algorithm. This diagram shows firstly, that this process only begins to take place after the Scheduler has started. This is due to the fact that prior to this point no threads will have been created. Following from the creation of the worker threads, the scheduler will enter into a looping mechanism that only exits when either the closing conditions are met or the abort flag has been set to true. Within this loop mechanism we can see that the scheduler iterates over the worker threads checking if they are sleeping or not, as a sleeping thread indicates that it requires further work.

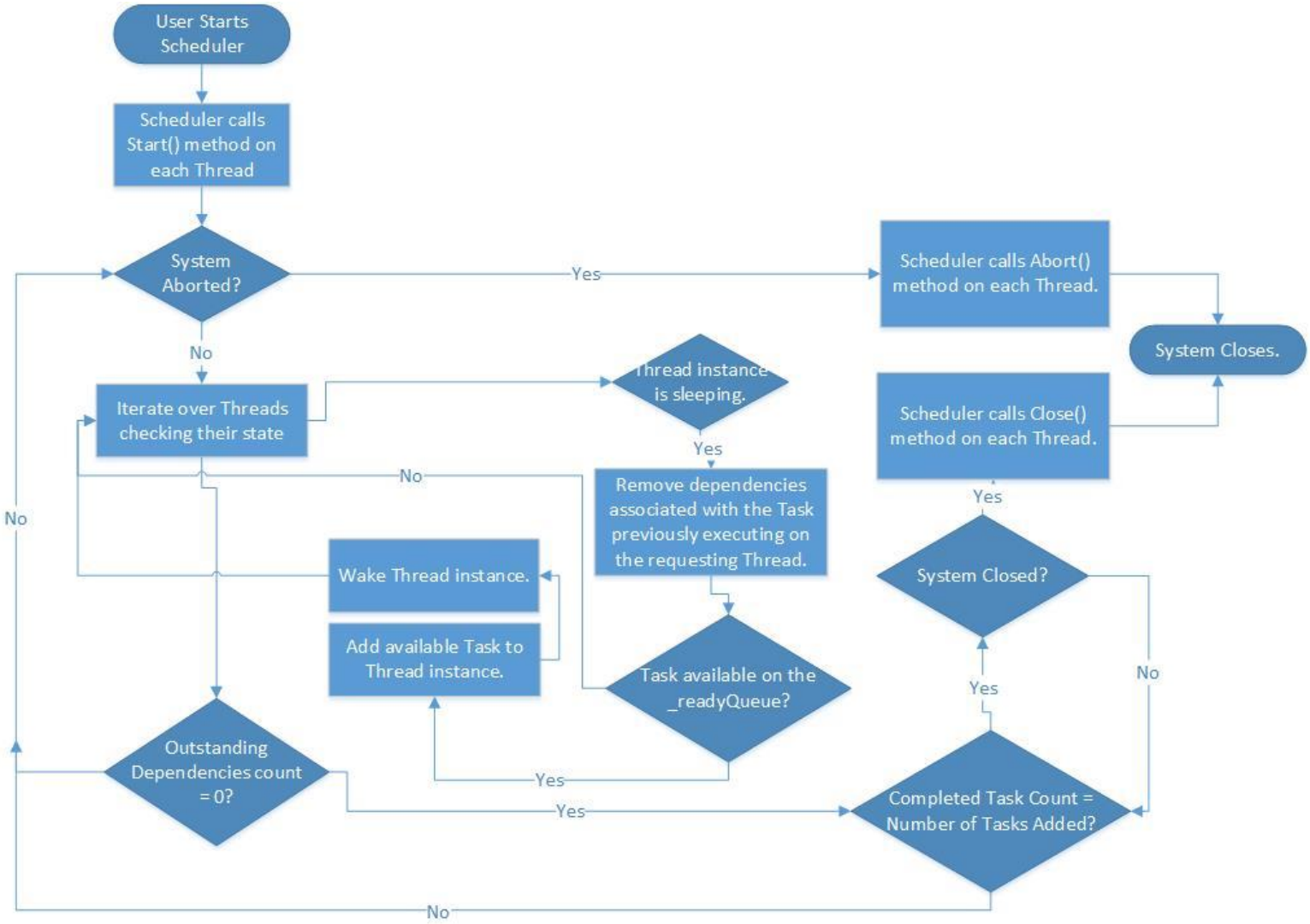


Figure 4 - Sharing Scheduler Task Assignment

If a thread is sleeping then we initially update the *_outstandingDependencies* list by removing any dependencies to the Task that was previously executing upon the thread in question, which can potentially lead to Tasks being moved to the *_readyQueue*. Following this the scheduler will check to see if the *_readyQueue* has any Tasks ready to be executed. If this is the case, then a Task is removed from there and assigned to the thread instance which will wake the sleeping thread thereby allowing the Task to be executed on the thread.

The diagram below (Figure 5) details the sequence of events that will take place within the scheduler under the Work Stealing algorithm. As with the Work Sharing algorithm, this sequence will only begin after the scheduler has been started. When the Start method is called under this algorithm the contents of the *_readyQueue* will be assigned at random to the available worker threads.

Once the scheduler has begun, the scheduler also enters a looping mechanism that ends when the previously mentioned closing conditions have been met. The differentiating factor between the two schedulers, is how they handle the completion of Tasks and the updating of the *_outstandingDependencies* list. This is done by firstly requesting the ID's of the completed Tasks from the worker threads. Once this has completed, the dependencies of the Tasks within the *_outstandingDependencies* list are updated. If one of these Tasks subsequently has 0 dependencies, then it is assigned at random to one of the available worker threads.

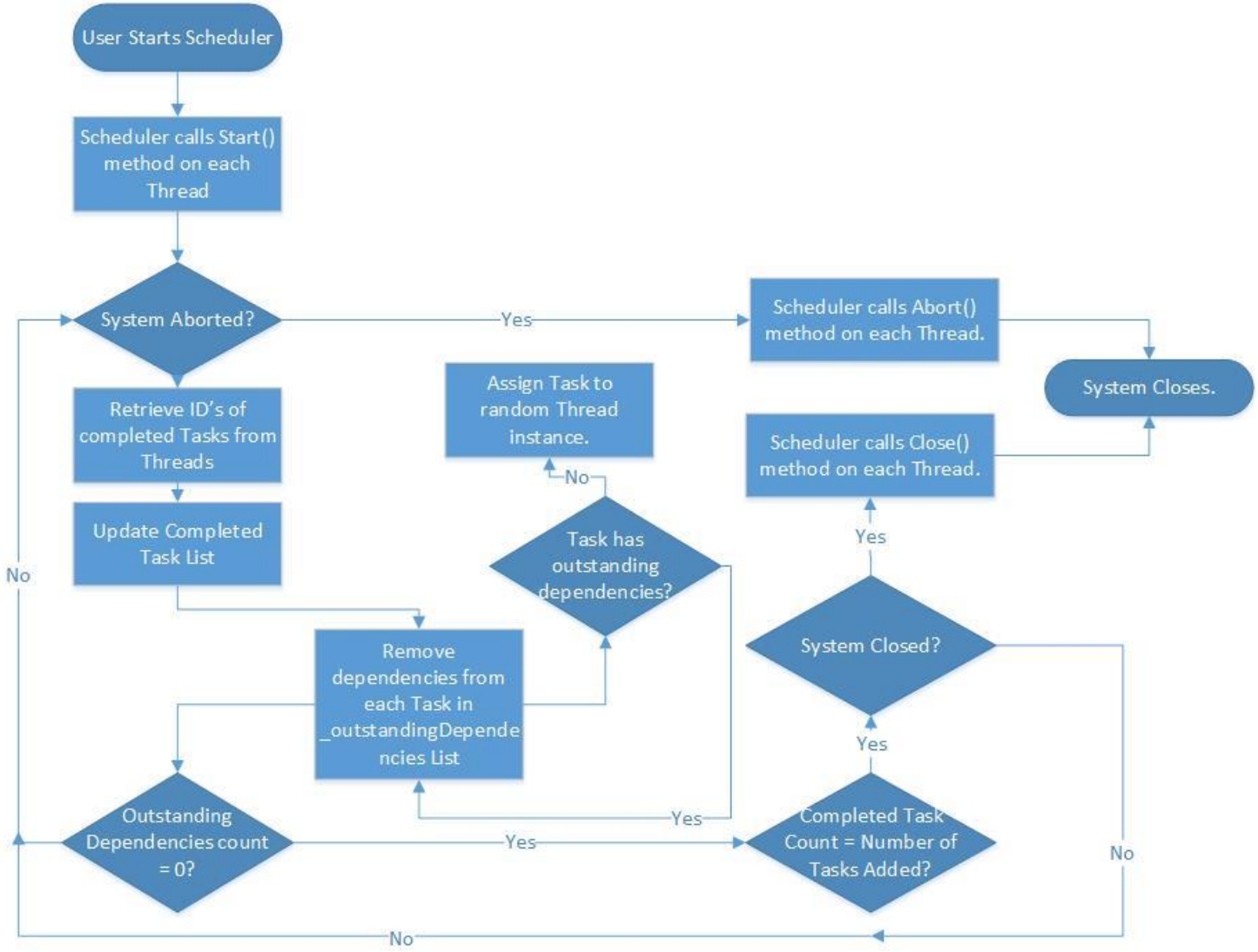


Figure 5 - Stealing Scheduler Task Assignment

4.1.1.2 Worker Thread

There is a large amount of cross over between the two different styles of Worker Thread, particularly with regards to the starting and closing functionality, however the point at which they differentiate is the functionality within the thread loop function. The thread loop, as its name implies, is a function that is executed by a C# thread which executes a sequence of commands in a loop until a stopping condition is met (such as when the Worker Thread is aborted or closed by the scheduler).

The thread loop of the Work Sharing Thread is relatively simplistic. The Thread itself can be in one of two states, executing and sleeping, and transitions between the two throughout its lifetime. If the Thread instance has a Task object available within its data store this will be executed by the Thread and it will enter the executing state. Otherwise the Thread will be sent to sleep so that it is not wasting processor time. The Thread will either be woken by the scheduler providing work, or after a period of time has surpassed such that the Thread can check to see if a closing condition has been met.

The thread loop of the Work Stealing Thread is similar in many respects, however it differs from the Work Sharing loop in how it handles a lack of work. Whereas the Work Sharing Thread will be sent to sleep, the Work Stealing Thread will follow the sequence of events detailed in the diagram below.

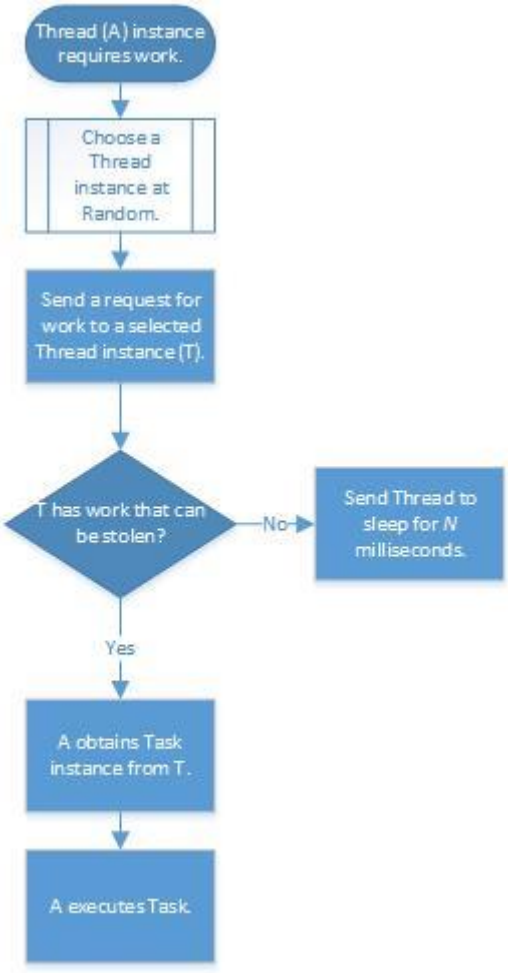


Figure 6 - Work Stealing Sequence Diagram

Figure 6 shows that when a Work Stealing Thread becomes underutilised, the thread will try to resolve this itself by attempting to “steal” Tasks from the deque of one of the other thread instances executing within the current application domain. If this process is successful, then the thread will execute the “stolen” Task object. However if this is not the case, the Thread object will be sent to sleep for a period of time.

4.1.1.3 Deque

As was mentioned previously, the most appropriate means to implement this data structure for this project was through an open doubly linked list. There are four functions present within the deque data structure through which we can access both the front (head) and end (tail) of the structure: Push, Pop, Inject and Eject.

The first of these, *Push*, is the function that places an element at the front of the deque structure and its functionality can be described using the following diagram.

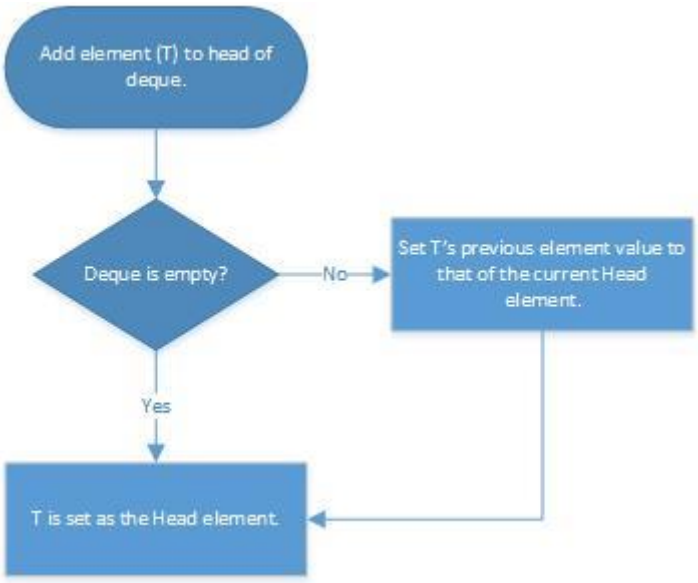


Figure 7 - Pushing an element onto the deque

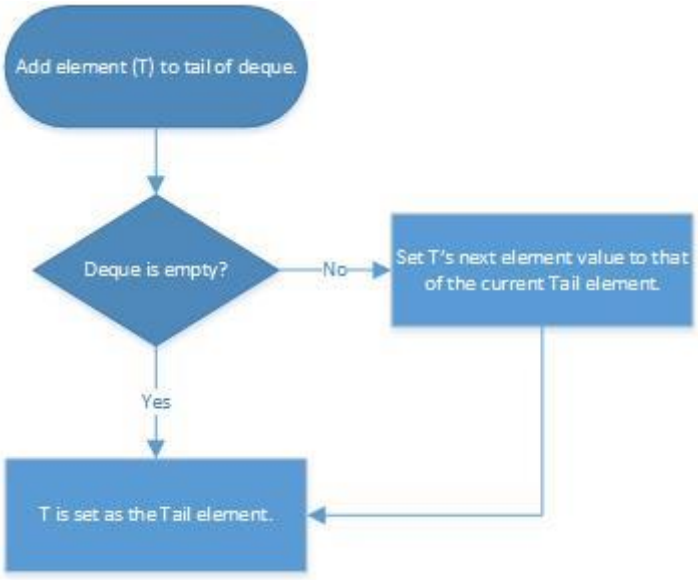


Figure 8 - Injecting an element into the deque

The *Inject* function is essentially the same function, but instead of placing the incoming element at the front or head of the deque it places at the end or tail.

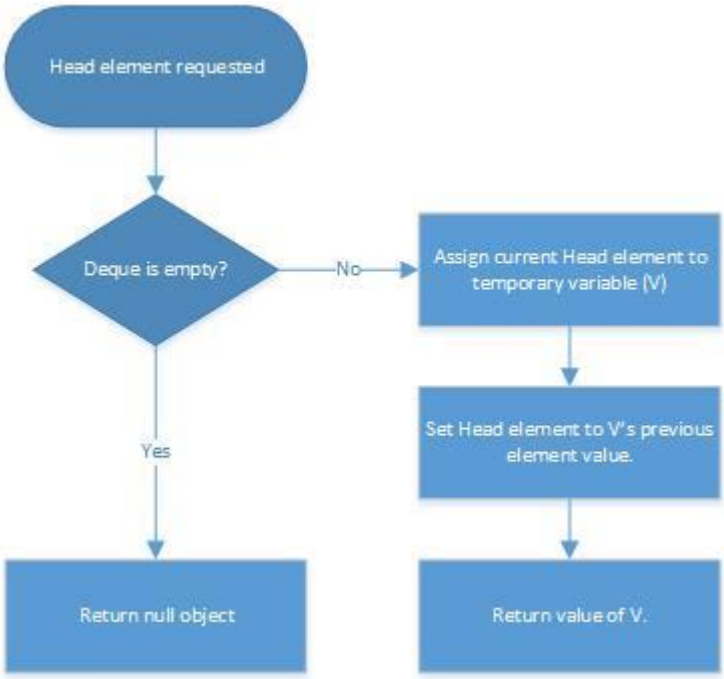


Figure 9 - Popping an element off the deque.

The *Pop* function (above) is the process through which the front most element is returned to the calling system. The *Eject* function, contrasts this by returning the element located at the tail of the deque instead.

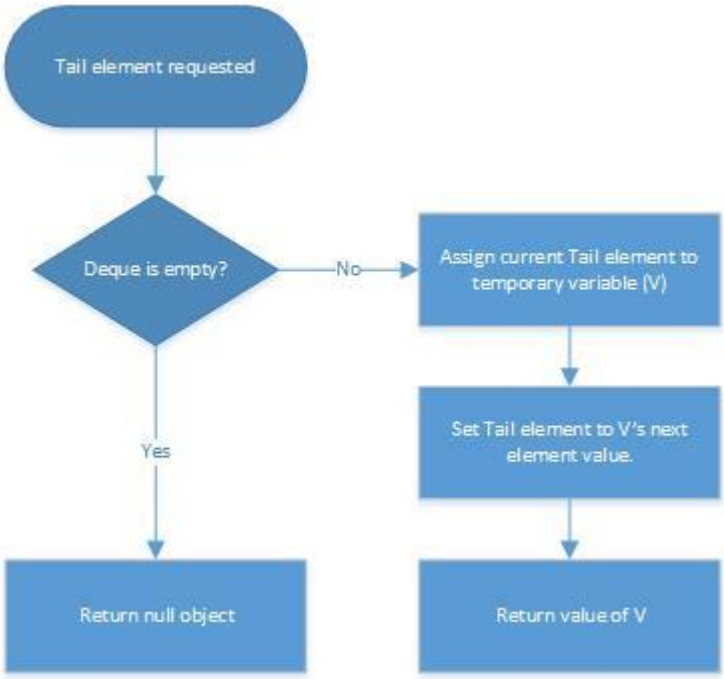
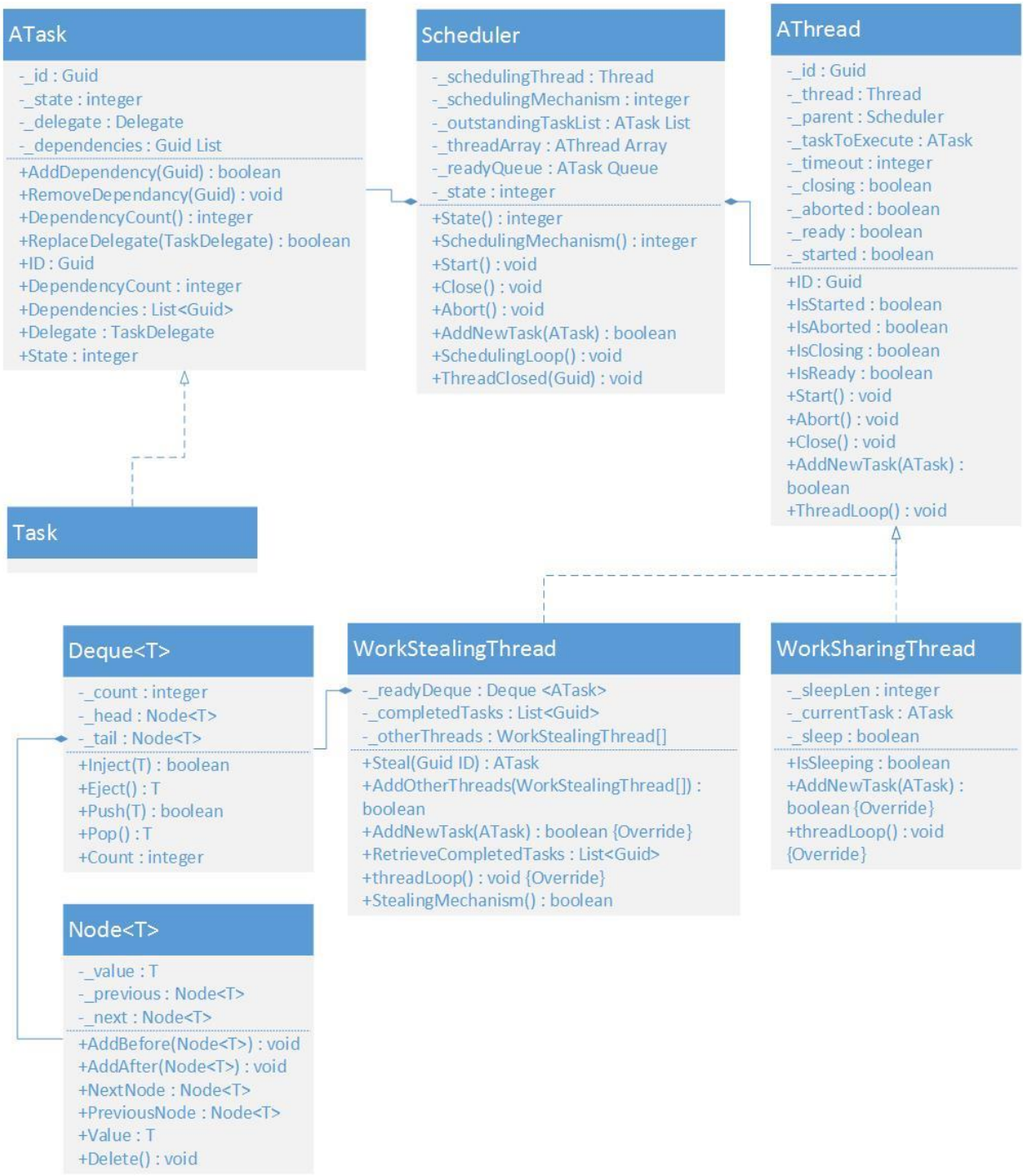


Figure 10 - Ejecting an element from the deque

4.1.1.4 Task

The Tasks that execute within the system are little more than wrappers functions which encapsulate the code that is to be executed by the Worker Threads. As such, the Task object itself is very simplistic and requires very little functionality other than a few accessor and mutator methods that either return or modify the data held within the Task itself.

4.1.2 Class Diagram



4.2 Test Design

Due to the complexity of the final system, it was necessary to use both unit and integration testing to ensure the system was fully functional.

4.2.1 Unit Testing

Given that the final system is comprised of several distinct interconnected classes, the use of unit testing within this system is appropriate as it allowed for an individual class to have all of its functionality checked under a variety of conditions prior to being combined with other classes. This therefore limits the chance of simple bugs within one class creating much larger issues when combined with other classes.

To ensure full coverage it was necessary to produce at least two test cases for each individual method within a class, although some methods had many more than this. Table 1, below, details a small number of the unit tests that were performed on the Deque object and their success.

ID	Name	Test Details	Success
1	AddToHead	Attempt to add valid T object to head of deque using Push(T) method.	True
2	AddToTail	Attempt to add valid T object to head of deque using Inject(T) method.	True
3	PopEmptyDeque	Attempt to remove an item from the head of an empty deque using the Pop() method.	True
4	EjectEmptyDeque	Attempt to remove an item from the head of an empty deque using the Eject() method.	True
5	PushGreaterThanCapacity	Attempt to add a valid T object to the head of a deque object that is at capacity. (See if it increases capacity)	True

Table 2 - Unit Tests for Deque Object

Of the 97 unit tests that were produced for this system, 97 produced the expected outcome.

4.2.2 Integration Testing

As unit testing can only provide us with the outcome of methods in isolation, it was also necessary for the system to undergo integration testing. By utilising this testing mechanism, it was possible for us to ensure that combined solution functioned as expected. With that being said the majority of the testing for the system was performed using unit tests, as it was much more feasible to produce exhaustive tests for the individual components than it was for all possible combinations of the components.

The integration tests comprise of four distinct tests which are designed to cover the various types of Tasks that can be passed to the system to be executed. The first of these tests, was very simplistic as the Task that was being executed was a lambda expression (Microsoft, 2015 b) that printed a predetermined string to the console. This first approach showed that both scheduling algorithms were capable of passing simple Tasks off to Worker Threads allowing them to subsequently be executed.

The second series of tests saw custom objects being passed into the scheduler, the objects in question were instances of a message class that can encrypt and decrypt a text string using a substitution cypher. These objects, were used to create a new Task by using one of the objects internal methods (Decrypt) as the function that is to be executed by the system. The data within the object had previously been encrypted during the creation of the objects. This test proved that both algorithms are capable of handling not just simplistic lambda functions but also functions from more complex objects.

The third test saw us build upon the previous tests by once again using the message objects, however for this test we would pass both the Encrypt and Decrypt methods in as separate Tasks with the latter being executed only after the former had completed execution. This was done by adding a dependency to the Task holding the Decrypt method, which referenced the Task holding the Encrypt method in the form of a copy of the GUID associated with this Task. By performing this test we were able to ensure that Tasks which are dependent upon others can be executed in the correct sequence.

The fourth and final test follows a similar vain to the previous, in that it is related to Tasks being dependent on one other Tasks, however this test is related to the creation of new Tasks within a Task. This is possible because the scheduler as well as the Task objects themselves are visible to the Worker Threads, which therefore allows the Worker Threads to create and pass Task objects to the scheduler when invoking another Task. To perform this test it was necessary to produce two simple methods, both of which simply print a string to the console, the first of these methods also creates a new Task with the second method being the associated function. Once this Task had been created, the AddNewTask method within the scheduler will be called from the first method which then added the new Task to the scheduler. By performing this test it proved that it is possible to employ subtasking within the system which is beneficial to the user as it means that dependencies do not need to be explicitly defined.

4.3 Experiment Design

In order to fulfil both objectives 4 and 5, it was necessary to design an experiment that enabled us to test and measure the system under a variety of conditions to determine if the system provided effective utilisation of the processor, as well as which of the two scheduling algorithms provided superior performance.

In order to test the system under a variety of conditions, a mixture of edge cases and standard parameters were implemented. An edge case, is an input that is at the extremes of the operating parameters allowed by the system (Zimmerman, 2013). For the system that was implemented, five edge cases were identified which are detailed in the table below.

Name	Operating Parameters
Heavy Load	Large number of Tasks requiring processing.
Limited Load	Small number of Tasks requiring processing.
Exception Handling	Tasks that could throw exceptions require handling.
Maximal Time	Computationally expensive Tasks need handling.
Minimal Time	Computationally inexpensive Tasks need handling.

Table 3 - Edge Cases

It was decided that the experiment would be based around pathfinding within a grid, as it would allow for all of these operating conditions to be fulfilled as well as being a real world usage of task based parallelism. See Appendix A for background information on pathfinding.

The experiment was such that it would utilise a class, called Grid, which would perform four functions. The Grid itself contains a list of *N* instances of another class called Square, which is analogous to a node object within a mathematical graph. The Square class itself holds the X & Y coordinates of the Square itself, as well as its size and a list of the Squares that are its neighbours.

The first function of the Grid class, was to read in a text file that represents a grid and replicate the grid based on the values within this file. Figure 12 is an example of the file format used. The first line represents the coordinates in the grid of the starting point. The second set of coordinates, after the # symbol, represents the end nodes coordinates. The rest of the coordinates represent nodes in the grid which are blocked, which should be navigated around by the pathfinding algorithms.

These files were generated by a custom piece of software which allowed for a graphical grid to be produced and converted into this style.

```
35, 35
#
985, 985
#
535, 35
535, 85
535, 135
535, 185
```

Figure 12 - Example of a Grid in text form.

The second and third functions, were the pathfinding algorithms. For this experiment two algorithms were implemented, specifically Dijkstras and A*, in an effort to produce more Tasks for the system to execute. See Appendix B for details on the implementation of these algorithms. It is worth mentioning here that Dijkstras algorithm was implemented using a Min-priority queue, due to the superior performance afforded by using this approach. Upon completion the algorithms will store the sequence of Square objects that they pass through within a list inside the Grid object.

The final function of the Grid class, is to produce a visual output of the Grid and the two routes produced by the pathfinding algorithms. This is done by the production of Bitmap file which

holds the grid, with squares within this grid highlighted in different colours if they were squares used to traverse from the start node to the end by the different algorithms. Figure 13 below, is an example of the output produced by this function.

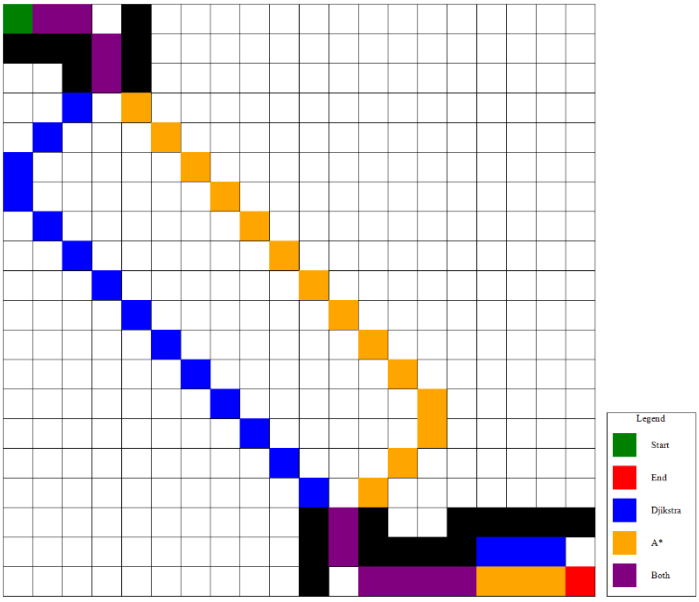


Figure 13 - Example output file.

4.3.1 Measurement Design

As well as designing the tests that were to be performed, it was also necessary for us to design the means through which we would measure the system during these tests. For this system, the measurements were recorded by a separate class that was instantiated within the Worker Thread object when it was created. The measurements were taken and recorded within the Worker Thread firstly as this is where the majority of the functionality occurs, as well as the fact that storing the data locally within a multithreaded system ensures that locks are not required to access and modify the data. The measurements that were taken are described in the following table:

Name	Description	Sharing	Stealing
Lifetime	The amount of time that the Worker Thread instance was active and executing tasks.	Yes	Yes
Throughput	The number of tasks executed by the Worker Thread during its lifetime.	Yes	Yes
Time Not Executing	The amount of time spent not executing tasks.	Yes	Yes
Average Execution Time	The average time taken to complete execution of a task.	Yes	Yes
Steal Attempts	Number of attempts made to steal tasks from other Worker Threads.	Yes	No
Successful Steals	Number of successful attempts to steal from other Worker Threads.	Yes	No

Table 1 - Description of measurements used.

5 Technical Implementation

Throughout this project, it was necessary to implement mutual exclusion through the use of exclusive locks. According to Microsoft (2016 b) the best practice for use of the lock statement, is to define a private instance of the Object class to lock on to. Within the system each class instance implements this, with the caveat that the object is specified as read-only thereby ensuring that this value is unchanging for the lifetime of the instance.

```
private readonly object _locker;
private Scheduler()
{
        _locker = new object();
...
```

Figure 14 - Instantiation of Object to lock on in Scheduler class

5.1 Task

As was mentioned previously, the Task class has a relatively simplistic design owing to the fact that it is little more than a wrapper for the function that is to be executed by the Worker Threads. With that being said there are few important features within this class that need to be discussed.

The first of these, is the mechanism through which the function that is to be executed is stored. The approach taken for this project was for the Task object to make use of delegates (Skeet, 2013). Delegates are means through which developers can define the 'signature' of a method, namely what its parameters and return values are, and then assign a method that matches this signature to an instance of that delegate. The delegate will then hold a reference to the method that was assigned to it. This approach is similar in many respects to function pointers found in both C & C++.

At present within the system there is only one delegate which has a 'signature' that has no parameters and does not return a value upon completion. During instantiation of each Task object, an instance of this delegate pointing to a specific function is passed as a parameter. This delegate is then stored within the Task object, which can then be accessed or modified through the use of specific functions.

```
static void Main(string[] args)
{
    //In this instance by passing a reference to doSomething() into the Task,
    //we are creating a new delegate that is then used by the Task object itself.
       Task test = new Task(doSomething);
}
public static void doSomething()
{
       Console.WriteLine("Hello");
}
```

Figure 15 - Passing delegate to a Task object

As the system was designed in such a way that allowed for interdependent Tasks to be added, a mechanism for storing and manipulating these dependencies was required. It was decided to perform these operations within the Task object itself firstly because it would be more user friendly, as the user would have all the functionality required within the class itself, and

secondly as it would mean that the scheduler would not need to access a complex data structure when updating the dependencies which could affect performance.

There were two approaches that could have been taken to store dependencies within the Task object. The first of which is to store the dependencies within a data structure of Task object. This approach has the benefit of being rather memory efficient as the Tasks would only be stored as object references which can be either 32 bits (4 bytes) or 64 bits (8 bytes) in length.

The second approach is for the Task object to contain a data structure of GUIDs, or globally unique identifiers, that are the GUID's of the Tasks that the Task is dependent upon. As with the previous approach the actual GUID values, would not be stored within the data structure but rather references would be held in the data structure. There is therefore very little difference between the two approaches in terms of memory or time trade-offs, and as such the approach that was taken was the latter of the two owing to the fact that it would be easier to differentiate between GUID's during debugging than Task objects.

To implement this within the framework, each instance of the Task class contains a List data structure which holds the GUID's of Tasks that the Task instance is dependent upon. A List was chosen instead of an array for this task, due to the fact that resizing operations for Lists are less computationally expensive than for arrays and several of these operations could take place over the lifetime of the Task.

Finally as the scheduler within this system is effectively the boundary between the user and the system itself, it was necessary to implement a mechanism through which we could limit access to the Task object once it has entered the scheduler. This was necessary to prevent accidental, or intentional, modification to the Task which could cause the system to crash if not handled correctly. To implement this mechanism, a simple Boolean value was utilised which is modified by the scheduler upon entry. This Boolean value is checked within each user accessible method and if it is set to true then the operation is refused.

5.2 Scheduler

As was detailed within section 4.1.1.1, the scheduler within this system implements the Singleton design pattern. Figure 16, below, specifies how this was implemented within the scheduler.

```
private static readonly Scheduler _instance = new Scheduler();
public static Scheduler Instance
{
	get
	{
		return _instance;
	}
}
static Scheduler()
{
}
private Scheduler()
{
	_locker = new object();
	_maxThreads = 8;
	_state = SchedulerState.NotStarted;
	_mechanism = SchedulingMechanism.WorkSharing;
	_readyQueue = new Queue<ATask>();
	_outstandingDependenciesList = new List<ATask>();
	_taskCount = 0;
	_threadString = new string[_maxThreads];
}
```

Figure 16 - Singleton Implementation

The reasoning behind this approach, is that it has the benefit of being thread safe without the need to implement any locking mechanisms. The reason that it is thread safe is due in part to the presence of a static constructor, which in C# is called when an instance of the class is created or a static member is referenced and can only execute once per AppDomain. This property is what ensures that only one instance of the scheduler class is ever created, even within a threaded application. The way in which the above code functions, is that when a user calls the Instance property for the first time, the static constructor will be called which creates a static instance of the Scheduler thereby making all of the static fields available. As the *_instance* variable is itself a static member, calling the static constructor leads to the non-static constructor being called which creates an instance of the scheduler class.

The Start function is another area of interest within the scheduler owing to the fact that it needs to have the capability to start two different executing from within one class. When this function is initially called the first task that is performed is that a call to the State property is made. This is done to ensure that the Start function has not already been called, which would affect the performance of the system due to there being a surplus of threads executing within the system. If the scheduler has a state of *NotStarted*, then we continue to start the functionality that is associated with the scheduler and worker threads.

This consists firstly, of changing the state of the scheduler and assigning and creating variables that will be utilised by the worker threads and the scheduler from that point forward. The main bulk of the work for the starting mechanism is done within a switch statement to initialise the worker threads. Within the cases for this switch statement, we first initialise the array of AThreads held within the scheduler to either WorkStealing or WorkSharing Threads depending on the algorithm being used. This array is then populated with new instances of the specific worker thread that is required. The system will then assign the scheduling thread a method which employs the appropriate scheduling algorithm in a loop until the closing

function is reached. It is worth mentioning here that if the algorithm is that of work stealing, then the Tasks within the ready queue will be assigned at random to the newly created worker threads. Following from this, both the scheduling thread and the worker threads will be started.

```
public SchedulerState State
{
	get
	{
		SchedulerState retVal = SchedulerState.Running;
		lock (_locker)
		{
			retVal = _state;
		}
		return retVal;
	}
}
```

Figure 17 - Thread Safe State Property

Figure 17 above shows the functionality of the State property, which is one of the many properties within the scheduler and the system as a whole. As was mentioned previously, this is an example of a thread safe property, thread safety is achieved here through the use of an exclusive lock to access the value of the class level variable before assigning its value to a local variable. This practice of locking the global variable then assigning it to a local variable is used in all of the properties within the system.

The Close function differs greatly to Start, as it simply modifies the scheduler's current state value to that of closing. The majority of the closing mechanism can be found within the scheduling loop methods.

The scheduling loop for the work sharing algorithm, is where the bulk of the work is done for this particular algorithm. Within this method we are firstly checking the state of each worker thread to determine if it is sleeping or not, which indicates a lack of work. If this is the case, we firstly update the *_outstandingDependencies* list based upon the Task that was previously executing on the worker thread. Once this is done we then dequeue a Task object from the *_readyQueue*, if there are Tasks available within the queue. The scheduler then adds this Task to the work sharing thread instance, after which the GUID of the Task is stored within an array that holds the GUID's of the Tasks currently executing.

The scheduling loop for work stealing differs to this approach as rather than having an internal array of the currently executing Tasks within the server, it requests a list of GUID's of the Tasks that have executed upon a specified work stealing thread through their *RetrieveCompletedTasks* method. Once the scheduler has retrieved these values for all worker threads the *_outstandingDependencies* list is updated accordingly. If a Task's dependency count reaches 0 during this, then it is assigned to a thread at random.

Within both of these methods is the functionality to close the scheduler and the associated worker threads. This is only called if several variables within the scheduler are set to the correct value. The first of these is to ensure that the scheduler has a state of *Closing*. If the scheduling algorithm is work sharing then we must subsequently check to see if the *_readyQueue* is empty, therefore indicating a lack of Tasks that need to be scheduled. Following from this we check the number of Tasks in the *_outstandingDependencies* list for the same reason. Finally, we check to see if the number of Tasks that have completed execution is equivalent to the number of Tasks that were passed into the scheduler. If this is true, then the closing method of the worker threads is called.

```
//Closing mechanism for the threads.
if (State == SchedulerState.Closing)
{
	if (ReadyQueueCount == 0)
	{
		if (OutstandingTaskCount == 0)
		{
			lock (_locker)
			{
				if (_taskCount == _completedTasks.Count)
				{
					closeThreads();
					break;
				}
			}
		}
	}
}
```

Figure 18 - Closing Mechanism for Work Sharing Algorithm

The final section of the scheduler, is the functionality related to adding new tasks. Within this method we firstly alter the boolean within the tasks that indicates that the Task has entered the schedule. Following from this we update the Tasks dependencies by removing the GUID's of any Tasks that have completed execution. If after this process the Task still has dependencies associated with it, then it will be added to the *_outstandingDependencies* list. Elsewise, the Task will be either stored or assigned depending upon the scheduling algorithm. If the work sharing algorithm is being used, then the Task will be added to *_readyQueue* to be assigned at the whim of the scheduling thread. However if the algorithm is work stealing, then we will either directly assign the Task to a random thread if the scheduler is executing or we will store them temporarily within the *_readyQueue* for assignment when the scheduler is started.

5.3 Worker Threads

The Work Sharing & Work Stealing Threads, although distinct from one another, have a large amount of shared functionality which is held within an abstract class called AThread. This class is comprised predominantly of properties which detail the current state of a worker thread. However some of the code within this abstract is worth mentioning, most notably the starting mechanism.

```
public bool Start()
{
        bool retVal = false;
        if (!IsStarted && IsReady) //Ensure Thread is not already executing.
        {
            _thread = new Thread(threadLoop);//Start new Thread on threadLoop method.
            _thread.Start();
            IsStarted = true;
            retVal = true;
        }
        return retVal;
}
```

Figure 19 - AThread Start Function

Although the above code may seem relatively simplistic, what makes it special is that once we have verified that the instance has not already been started we create and start a new Thread object which runs on an abstract method, *threadLoop* that is overridden within each class. By doing this, it means that we can have a common Start function that is applicable for any children that inherit from this class.

The AThread class also contains the abstract *AddNewTask* method. This is necessary as we need to use the same method within the scheduler to add tasks to the worker thread classes, however as the two worker threads differ in how they store Tasks we must override them within the child classes.

5.3.1 Work Sharing

The *AddNewTask* method within the Work Sharing Thread object, functions firstly by ensuring that the Thread instance is in the Sleeping state. This is necessary because a sleeping Work Sharing Thread indicates that the Thread instance has a lack of work, therefore adding a new Task to the instance would be appropriate. Once this is done, we check the validity of the incoming Task by first ensuring that the delegate of the incoming Task is not null and then checking that the State of the Task is that of *ToBeExecuted*. If all of these checks pass, then we first set the *_currentTask* variable to the Task being passed in. Following from this we use the *Interrupt* function on the *_thread* variable, which sends a ThreadInterruptedException which will wake the thread from its sleeping state.

Within the Work Sharing variant of the *threadLoop*, we have the main loop which checks to see if the Thread has been aborted or not. If this is not the case we then check to see if the Thread is closing or not, which if true causes the thread to break out of the encapsulating loop. We then check the Task that is stored in the *_currentTask* variable to ensure that it is still in an acceptable state and does not have a null value. If this is true then we assign the delegate value of the *_currentTask* to a local variable.

We then check that the local variable has value, after which we can then attempt execution. This is done by calling the delegate classes *Invoke* method, which executes the code encapsulated by the delegate within the calling thread. This is done within a try catch method to ensure that any exceptions that spawn when executing the delegate do not cause the entire system to crash. Following from the successful completion of the delegate, we set the state of

the Task that has been executed to that of *Completed* and nullify both the *localDelegate* and *_currentTask* variables. We then send the Thread instance to sleep for SLEEP_LEN microseconds, which is set to a constant value of 1000. Again this process is done within a try catch method, so that if a new Task is added to the Work Sharing Thread instance whilst it is sleeping then it will be able to handle the *ThreadInterruptedException* that is created when the *Interrupt* method is invoked.

5.3.2 Work Stealing

The *AddNewTask* method for the Work Stealing Thread, is very different from that of the Work Sharing Thread. The majority of the work within this function is focused around ensuring that the Task being passed in is valid, by checking its current state and that its delegate does not have a null value. Once these checks have been complete, we simply call the *Inject* method on the deque stored within the instance of the class. The *Inject* function, as detailed in section 4.1.1.3, is used to add the new Task to the tail end of the deque.

```
public override bool AddNewTask(ATask task)
{
	bool retVal = false;
	if (task != null)
	{
		//Ensure that the ATask instance has valid delegate to be executed
		//& has not been aborted or perviously executed.
		if (task.Delegate != null && task.State == TaskState.ToBeExecuted)
		{
			//Add ATask to the tail of the Deque.
			retVal = _deque.Inject(task);
		}
	}
	return retVal;
}
```

Figure 20 - AddNewTask method of Work Stealing Thread

The *threadLoop* of the Work Stealing Thread class is similar in many respects to that of the Work Sharing Thread. They differ on two points within this method, the first of which is related to where they get the next Task instance from. Whereas the Work Sharing Thread has its next Task assigned to it by the scheduler and stored in a single variable, the Work Stealing Thread stores any Task given to it within its internal deque data store. Therefore on each iteration of the *threadLoop* the Pop function is called, which returns the front most Task on the deque if there is a Task available otherwise null is returned. This value is then assigned to a local variable within the Work Stealing Thread instance. Following this process the checking and invocation of the delegate is identical to that of the Work Sharing Thread.

The second point at which the two Worker Threads differ is how they handle a lack of work. Whereas the Work Sharing Thread will go to sleep when it has executed is latest Task, the Work Stealing Thread will first attempt to get a Task object from the front of the deque. If this is unsuccessful, the Work Stealing Thread will then attempt to steal Tasks from other Threads through the *stealingMechanism* method. If this function does not result in a Task being added to the deque, then the thread will subsequently sleep for a period of time.

The way in which the *stealingMechanism* functions is that we firstly choose a random value, V, between 0 and N-1, where N represents the number of other Threads that are alive in the system. Once this value is obtained, we then access the Work Stealing Thread at address V within the array of other Threads held within the instance. If the instance at address V has not closed at the time of access, we subsequently make a call to its *Steal* method. This method

calls the Eject function on the internal deque of the encapsulating Thread instance, which will return either a null value or the Task located at the tail end of the deque. Once we have returned to the calling instance, we determine if the returned value is a valid Task or not. If this is the case then this Task is then injected at the tail of the deque data structure.

5.3.2.1 Deque

As was mentioned previously, this data structure has been implemented using an open doubly linked list. In order to implement this style of linked list, we must create a new class that can be used to represent a single node within the linked list.

The class that was created, Node<T>, utilises the generic capabilities available within C#. By doing this, it allowed us to design and implement a class that was type safe without explicitly declaring the type beforehand. The internals of the Node<T> class are relatively simplistic, as it is little more than a wrapper of the T value (where T is an ATask object within the system) with two object references to the previous and next Node<T> instances in the linked list. These objects references form the basis of the linked list, provide the 'links' between multiple nodes.

Due to the specifics of how a deque functions, namely that the data structure can be accessed from its head and tail, it is therefore necessary to store within the deque two instances of the Node<T> class where one is representative of the head and the other the tail. As was previously stated in section 4.1.1.3, the deque has four functions which are Pop, Push, Inject and Eject.

```
public bool Push(T value)
{
	bool retVal = false;
	Node<T> node = null;
	if (value != null)
	{
		lock (_locker)
		{
			if (_head == null)
			{
				node = new Node<T>(value, null, null);
				_head = node;
				_tail = node;
				++_count;
				retVal = true;
			}
			else
			{
				node = new Node<T>(value, _head, null);
				_head.AddAfter(node);
				_head = node;
				++_count;
				retVal = true;
			}
		}
	}
	return retVal;
}
```

Figure 21 - Push function of Deque

The *Push* function, adds a new T value to the front of the data structure. To do this, it firstly checks to ensure that the T value passed in is valid. If this is the case, we then acquire the deques internal lock so that we can access the private members of the deque in a thread safe

manner. Once the lock is acquired, we check to see if the *_head* Node<T> variable has value or not. If this is not the case then the incoming T value will subsequently become both the head and tail node of the linked list due to it being empty. If however there are instances of the Node<T> class within the linked list then the incoming T value will replace the current node at the head of the list. To do this we firstly create a new instance of the Node<T> class. After which we use the *AddAfter* function on the current head value to add the newly created node as the node ahead of this value in the linked list. Finally, we assign the *_head* variable the value of the new node to subsequently allow the head of the list to be accessed within the deque. The *Inject* function is effectively a mirror image of this function, with the only difference being that it adds the incoming T value to the tail of the deque instead of the head.

The *Pop* function, meanwhile removes and returns the T value that is located at the head of the linked list within the deque. This is achieved, by firstly ensuring that there is currently a value at the head of the list. If this is not the case, then we immediately return the default value of type T. If however there is a valid node at the head of the list, we set the return value to the current head node and assign a local Node<T> variable the value of the node that comes after the current head node within the linked list. Once this has completed, then the *_head* value will be assigned to the value held in the local variable after which we return from the method with the return value. As with *Push* and *Inject*, the *Eject* function is equivalent to the *Pop* function with the only difference being that the *Eject* removes the value from the tail of the linked list.

6 Experiment

6.1 System Configuration

The system on which these experiments were performed has the following specification:

- Intel i7-2600k CPU at 3.4 GHz
  - 4 Cores
  - 8 Hardware Threads
- 8GB DDR3 RAM
- Microsoft Windows 8.1

As well as this, the number of worker threads within the frameworks thread pool was set to a constant value of 8. Finally, for these experiments a job is broken down into 5 distinct Tasks, as detailed in section 4.3. For example, if an experiment executes 1000 jobs then this can be seen to mean 5000 Tasks will be executed.

6.2 Edge Cases

6.2.1 Heavy Load

To test for this edge case, it was necessary to execute a large number of instances of the aforementioned Grid class. For this experiment 5000 jobs were executed owing to the fact that although this is not the upper limit of the system, was large enough that it gave an example of a heavy load but within a time frame that was not excessive. A 30x30 grid was chosen for the fact that it could be executed within a reasonable time frame. If the grid had been larger, as we will see later, the time taken to execute would have been far higher.

Figure 22, shows that under a heavy load the work stealing algorithm completes execution in less than half of the time of its work sharing counterpart. If we also take into consideration the percentage of time that each thread is not directly executing a task, Figure 23, we can come to a conclusion as to why this is the case.

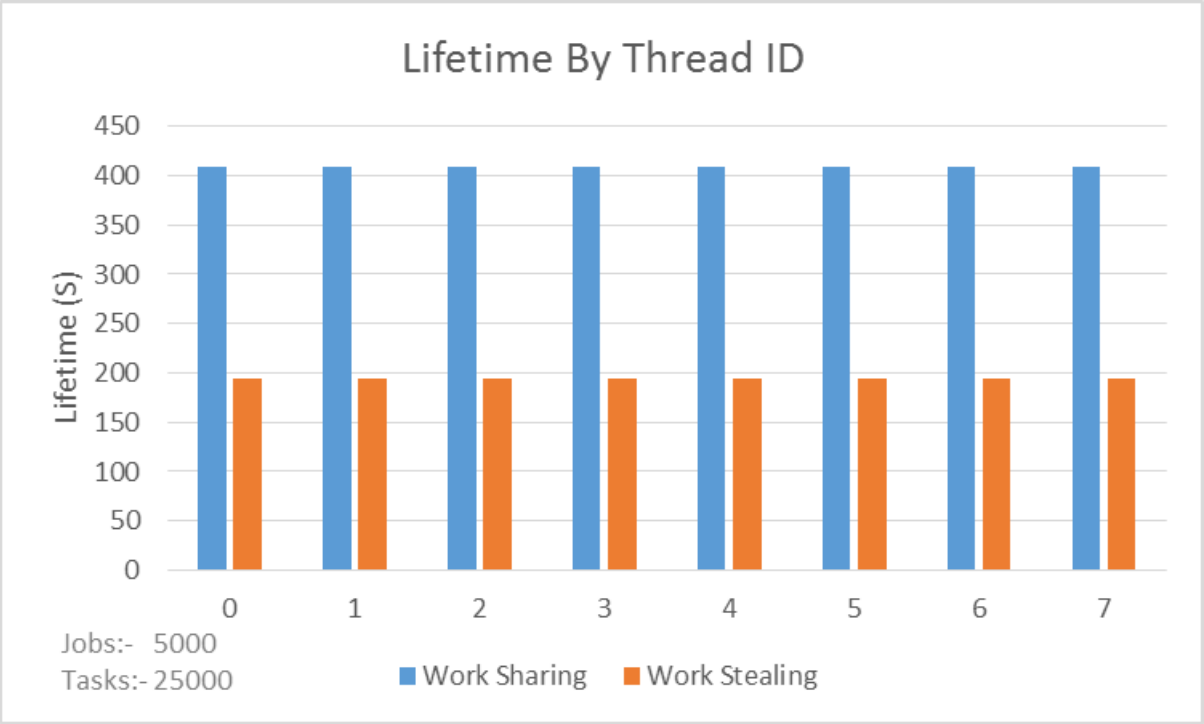


Figure 22 - Lifetime of Threads under Heavy Load

As can be seen, under a work sharing algorithm an average of 60% of each threads time is spent not directly executing Tasks. Whereas the work stealing algorithm spends on average less than 5% of its time not executing Tasks. The disparity between the two algorithms on this data set is due to the design of them, in that under work stealing the worker thread will actively attempt to find new work whilst the work sharing threads will go to sleep. Evidently when faced with a large number of Tasks, the work stealing algorithm would be the preferential choice.

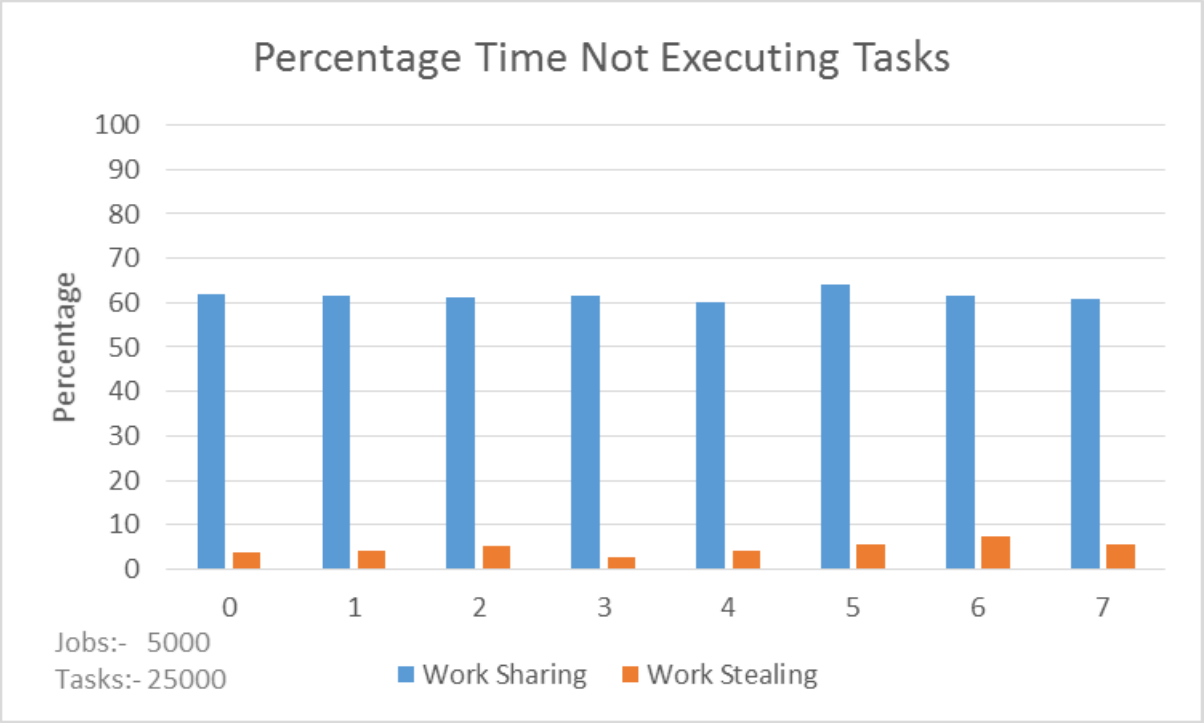


Figure 23 - Percentage of Time Spent Not Executing Tasks under Heavy Load

6.2.2 Limited Load

This edge case, can be seen as a polar opposite to the previous one. As such to test it is necessary for us to pass a very small number of jobs, specifically only 1, which means that at most 5 Tasks are executing on the system during its lifetime. A single instance was chosen in lieu of no instance to see how the system would handle having some worker threads undergo starvation. Once again a 30x30 grid was chosen such that it was consistent with the Heavy Load edge case.

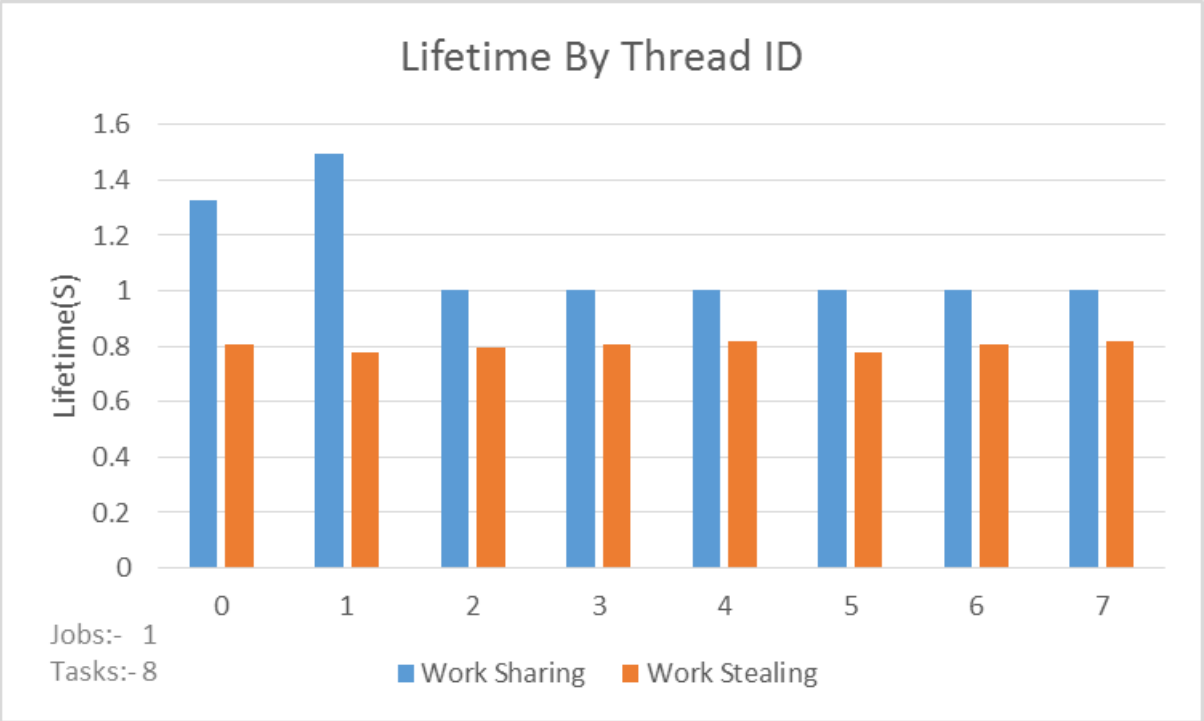


Figure 24 - Lifetime of Threads under Limited Load

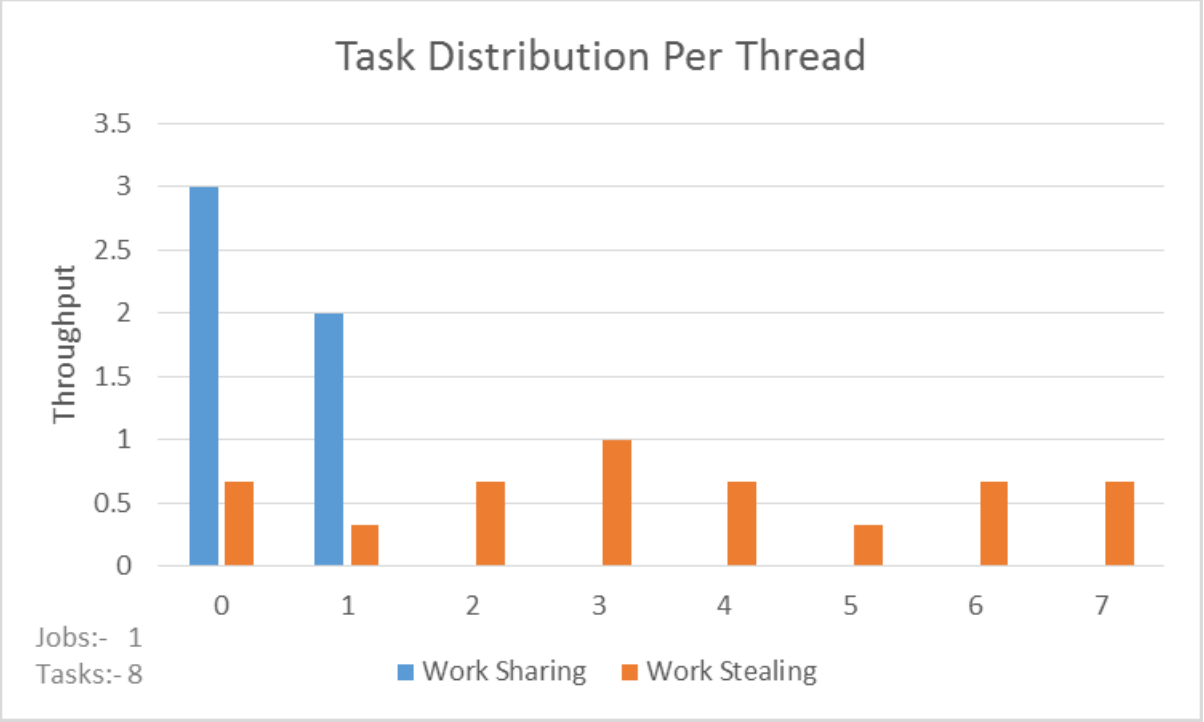


Figure 25 - Task Distribution to Threads under Limited Load

As can be seen from Figure 24, use of the Work Stealing algorithm results in quicker completion of the Tasks. Which is again owing to the fact that the sharing algorithm will sleep when underutilised. The interesting point to take away from this graph, are the spikes on threads 0 and 1 under the work sharing algorithm.

These peaks can be explained using Figure 25, which shows that under the work sharing algorithm only threads 0 & 1 ever receive any Tasks. This in turn causes the spikes in lifetime for these threads due to the fact that they must finish executing any Tasks before they can be closed unlike the other threads which can close as soon as the scheduler changes the close variable.

Given this data, it could be argued that when faced with a small load such as this that it would be advisable to again utilise the work stealing algorithm. However, given the overhead needed to create and assign to the worker threads performing these operations in parallel would not be the most efficient method.

6.2.3 Exception Handling

To test the exception handling for this system two tests were performed using 8 jobs each, the first of which was used to determine a base level of performance. To gain this information, the data that we passed into the first test sequence was data that was appropriate for the 10 x 10 grid that was being utilised. The second test, however, was used to gather the performance of the system when an exception is thrown when invoking the Task's delegate. To do this data that is inappropriate for a 10 x 10 grid is read into the system, which in the instance was data for a 20 x 20 grid.

Within the second test, three exceptions were thrown for each job (24 system wide). The first of these exceptions was thrown during the reading sequence due to the invalid data. The other two were thrown when the two pathfinding algorithms tried to execute on a null data set, which was caused by the previous exception ending the reading sequence early.

What we can take away from these two tests, is that the system that has been produced is capable of handling exceptions owing to the fact that at no point did the system crash due to an exception being thrown. However, the manner in which exceptions are currently handled is not particularly elegant and can as a result lead to a chain of exceptions being thrown. A solution to this would be to implement a system wherein any Task that is dependent upon a Task that has caused an exception would be removed from the system completely so that the number of exceptions encountered would be reduced.

An interesting point that came about as a result of these tests, is that the work sharing algorithm which had multiple exceptions was marginally faster than its normal counterpart. The potential reason for this, is that as exceptions occurred in the pathfinding algorithm functions no output list was produced for these algorithms and as such no graphical output was produced which could be time consuming.

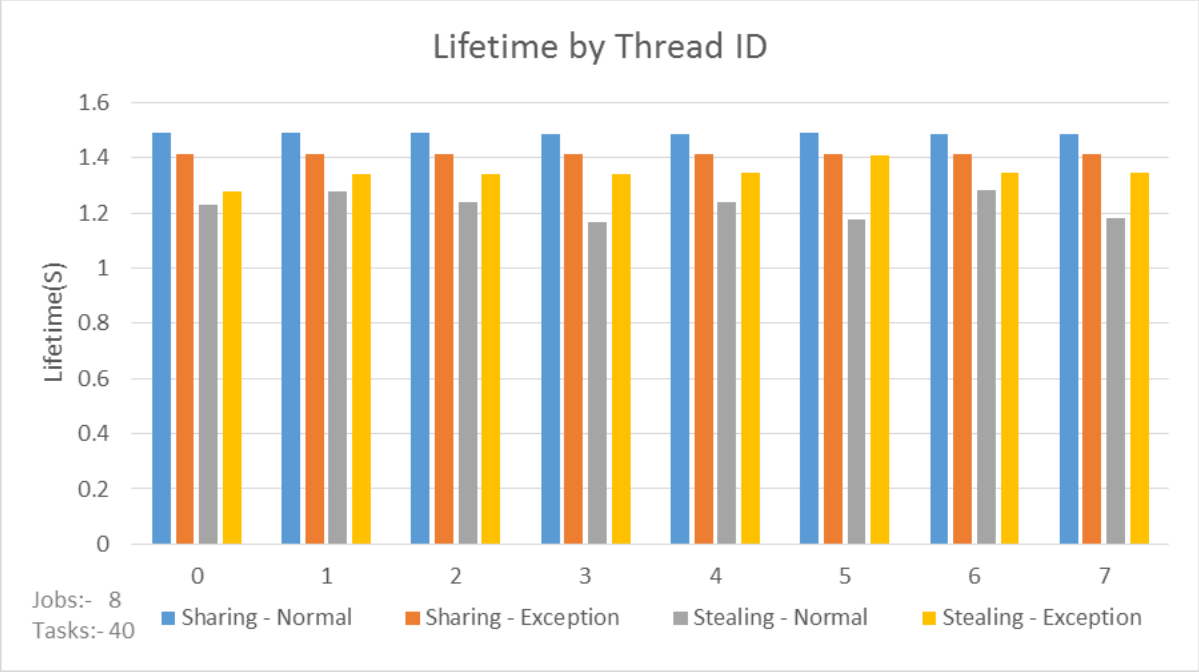


Figure 26 - Lifetime of Threads under normal and exception throwing circumstances.

6.2.4 Minimal Execution Time

This edge case was tested by performing 8 jobs on a 10 x 10 size grid. This size of grid was chosen owing to the fact that its limited number of edges and vertices, would result in a relatively short execution time for the both Dijkstra's algorithm and A* which are the two methods that require the most execution time due to their complex nature.

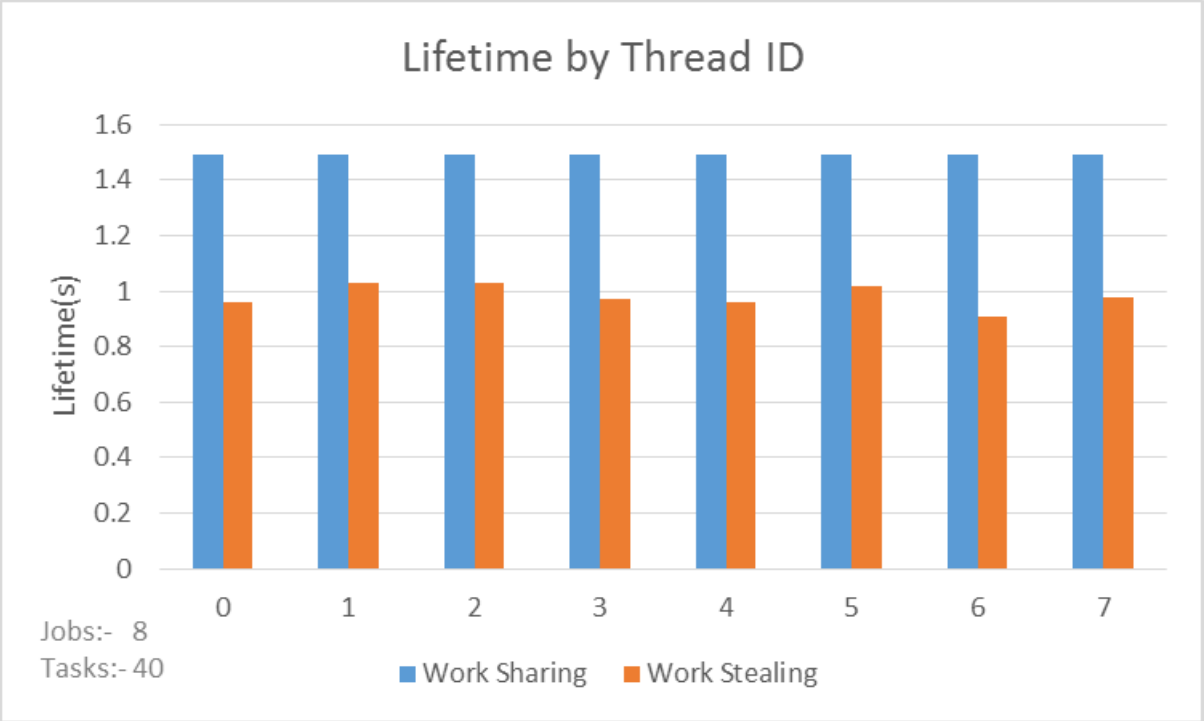


Figure 27 - Thread Lifetime when executing 8 Short Running Jobs

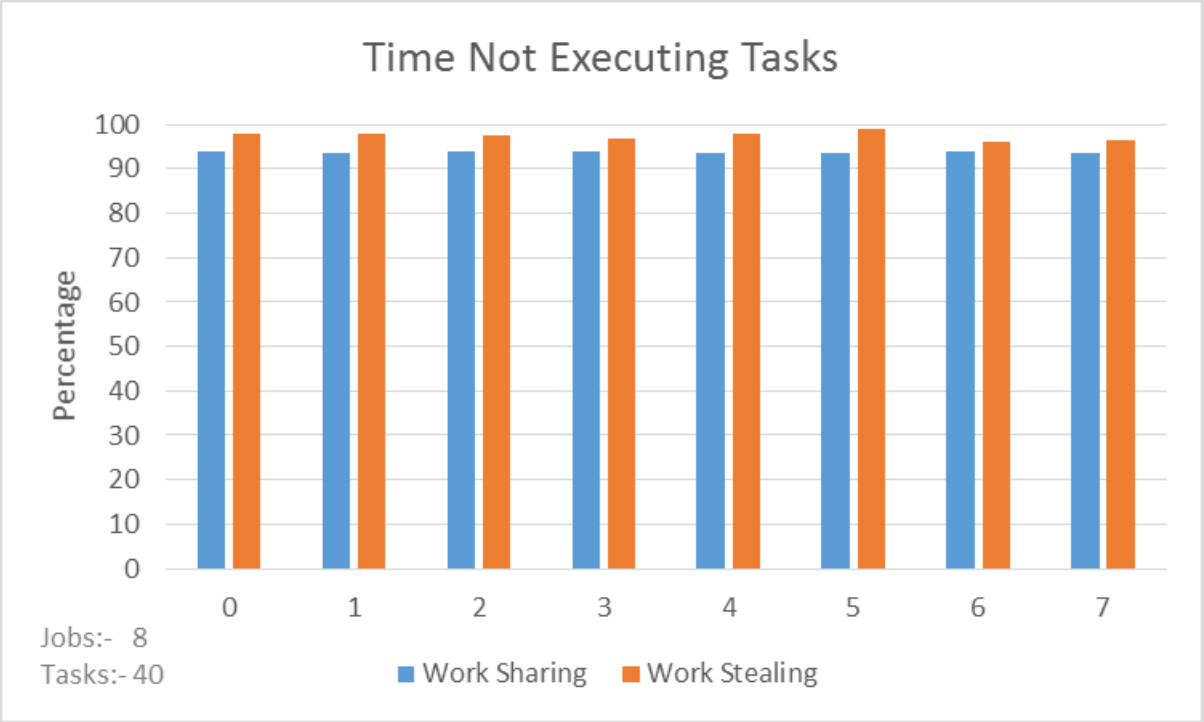


Figure 28 - Percentage of Time Not Executing Tasks with Short Running Tasks

What can be gathered from this test, is that once again work stealing is a superior algorithm in terms of execution time as can be seen in Figure 27. However, the work stealing algorithm also spent marginally more time not directly executing Tasks as can be seen in Figure 28. This is owing to the small number of Tasks that were being executed by the system, thereby inhibiting the stealing mechanism which will mean that the work stealing thread will be more inclined to sleep for a short period thus increasing the amount of time not spent directly executing Tasks.

6.2.5 Maximal Execution Time

To test for this edge case, eight grid objects were created with data that was for a 100 x 100 grid. This size of grid was chosen due to the fact that it provided an example of extremely long running Tasks, which could execute within a somewhat reasonable time frame. This increase in time is due to the nature of the pathfinding algorithms, in that an increase in the number of nodes within the search area will result in an increase in the time required to find a path within it.

As can be seen from Figure 29, when faced with a small number of long running Tasks the Work Sharing algorithm has superior performance to its stealing counterpart. As well as completing the Tasks in a shorter time frame, the Work Sharing algorithm also spends a greater percentage of its time directly executing the Tasks themselves. Finally, as Figure 30 displays, the percentage of successful steals for the Work Stealing algorithm under these conditions is low across the board.

Given this information, we can begin to draw a conclusion as to why the Work Sharing algorithm displays better performance characteristics. Firstly, as the number of Tasks added to the system is relatively low successfully stealing work form another worker thread would be rather difficult, due to a limited number of Tasks being available in other worker threads deques. As well as this, as the work stealing algorithm distributes Tasks in a random fashion it means that if the stealing mechanism is impeded by a small number of available Tasks then it can lead to some worker threads experiencing work starvation. Finally, as the majority of the Tasks for this test were reliant upon others completing before they themselves could be executed, it means that a much smaller number of Tasks were available at any one point.

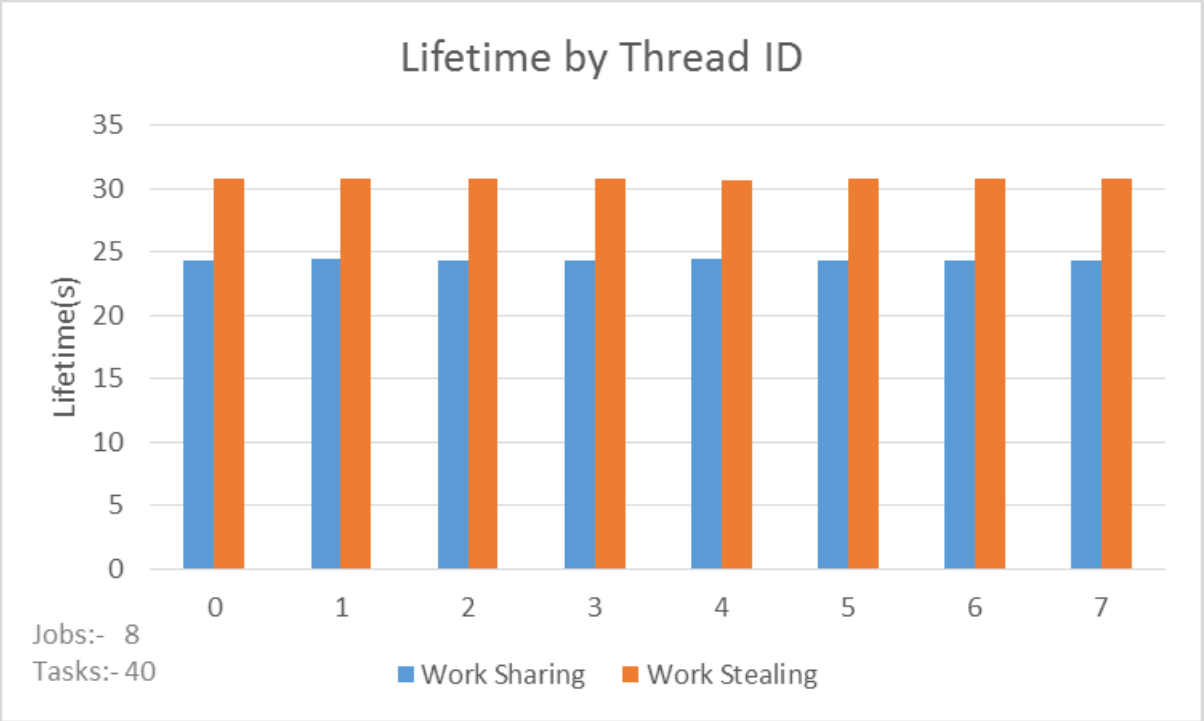


Figure 29 - Lifetime of Threads with Long Running Tasks

If we look directly at the data we can see that thread 0 had a somewhat low throughput of 2 Tasks as well as having made almost double the number of steal requests of which only ~16% were successful. We also see that this thread instance wasted a much higher percentage of its execution time, when compared with the other worker thread instances, as it was not directly executing Tasks. This therefore indicates that this thread experienced some form of starvation and was subsequently not utilised as effectively as it could have been.

The Work Sharing algorithm, however, provides a much more even distribution of Tasks to threads. As such, this means that less time is spent not executing Tasks when compared to the work stealing algorithm. As such, the conclusion that can be drawn is that if we wish to execute a small number of Tasks that are long running, then the Work Sharing algorithm will provide superior performance.

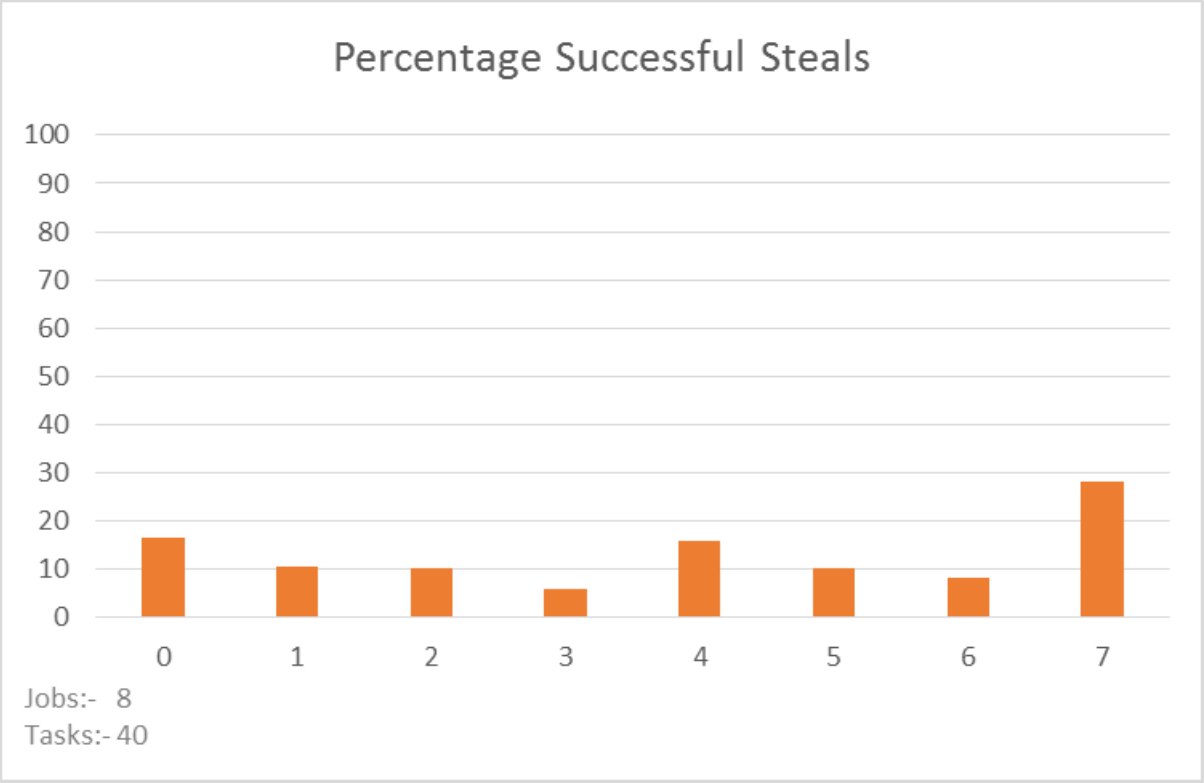


Figure 30 - Percentage of Successful Steals when Executing Long Running Tasks

6.3 Standard Operating Conditions

6.3.1 Dependencies Vs Subtask

Within the framework, it is possible to create Task objects that are dependent upon other Task objects in two distinct ways: through the dependencies mechanism within the framework and through the generation and addition of new Tasks within the code of the Task that is to be executed (subtasking). Owing to this, it was deemed to be beneficial to determine which of these two mechanisms provided superior performance for interdependent Tasks.

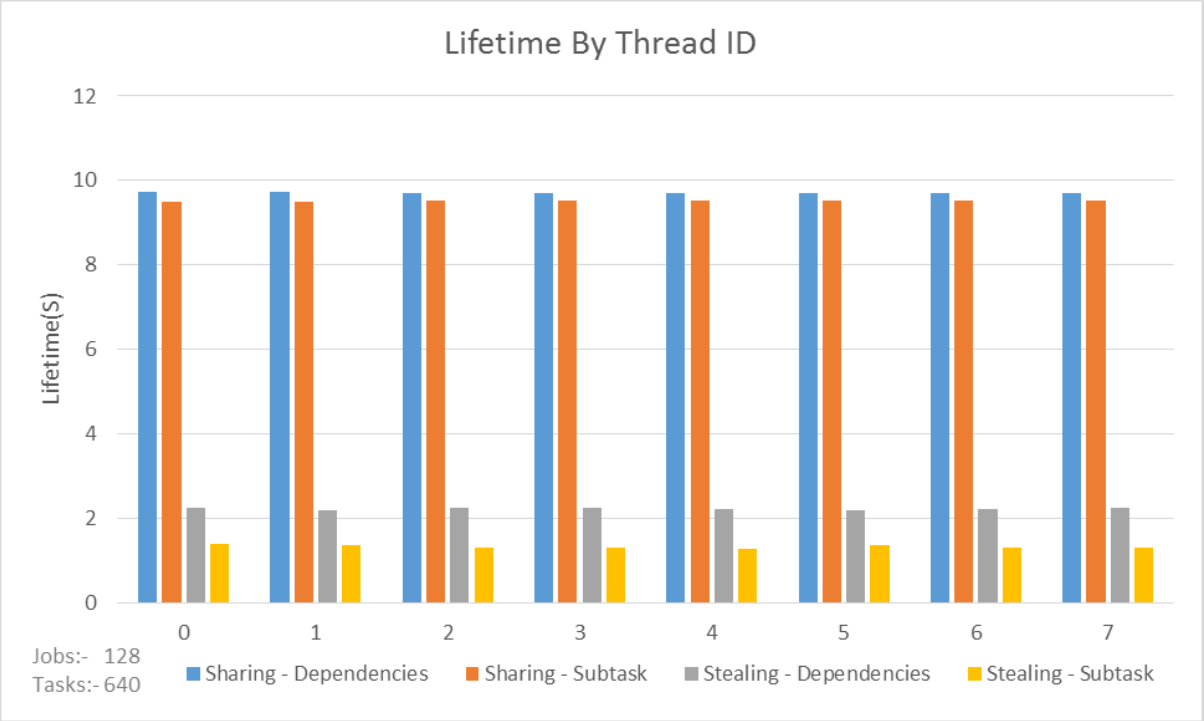


Figure 31 - Lifetime of Threads executing 128 Grids

The means through which this was tested was by passing *N* jobs into the system, with data from a 30 x 30 array. These grids were passed into the scheduler in four styles: sharing with dependencies, sharing with subtasking, stealing with dependencies and stealing with subtasking. Initially 8 jobs were utilised, however as this provided relatively inconclusive results this value was increased to 128.

As can be seen from Figure 31, utilising subtasking under the work stealing algorithm enables you to complete execution in the shortest period of time. However, this graph doesn't show the entire story, for which we need Figure 32.

What this graph details is that when using a Work Stealing algorithm and subtasking, less than half of the Tasks that are passed in will be executed. The reason that this is due in part to the nature of the Work Stealing algorithm, in that it will attempt to retrieve work from the tail end of another worker threads deque. This therefore means that Tasks are not necessarily performed in the order that they are added to the scheduler, which could mean that the *Close* function could be called before all of the other Tasks have been completed. The fact that Tasks that are generated using subtasking are only created when executing the Task that comes before them, and that closing the scheduler is dependent upon having completed all the Tasks that have been added to the scheduler until that point enables the system to exit before all of the Tasks have been generated.

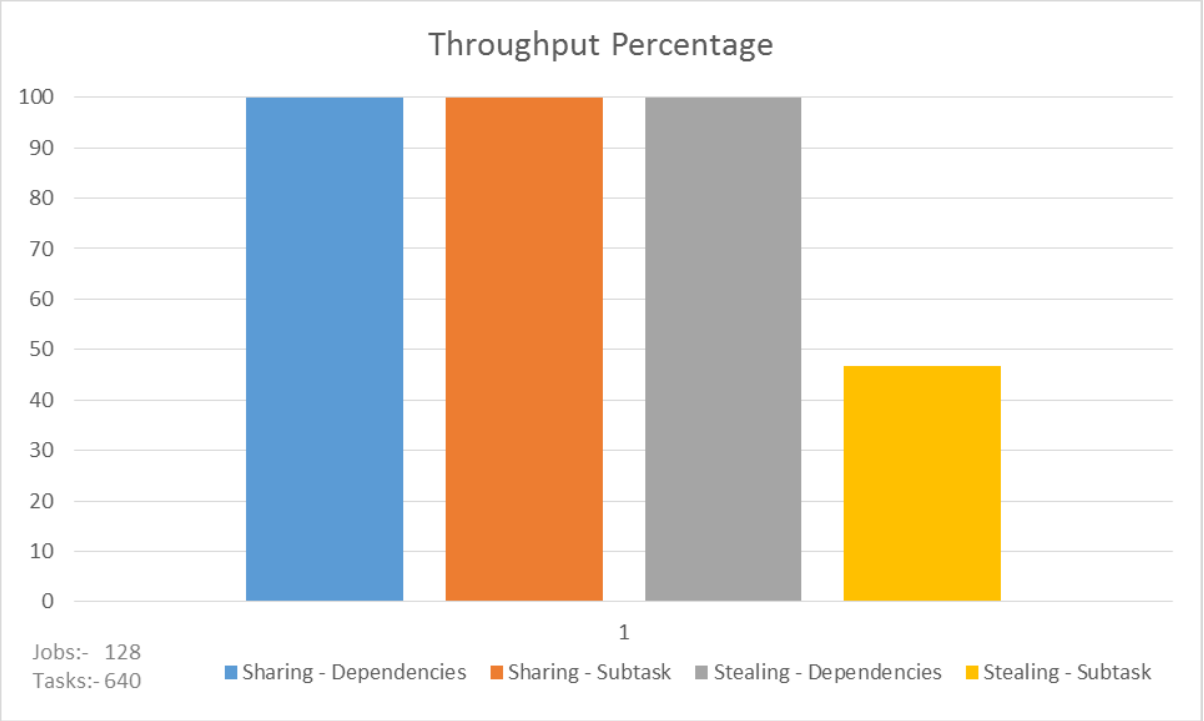


Figure 32 - Throughput percentage of 128 Grids

With this in mind, we can therefore state that the best mechanism for execution of Tasks in a timely manner is to use the Work Stealing algorithm and the built in dependencies of the framework. It is also worth noting that if under a work sharing algorithm, utilising subtasking allows for the Tasks to complete marginally faster.

6.3.2 Varied Job Count

The aim of this test, was to see how the scheduling algorithms that had been implemented compare across a variety of number of jobs that have to be completed. To do this *N* jobs were passed into the system, which execute upon a 30 x 30 grid.

As well as comparing the algorithms against one another, they were also compared with the same jobs being executed in a non-parallelised manner. The following graph displays the results of this test.

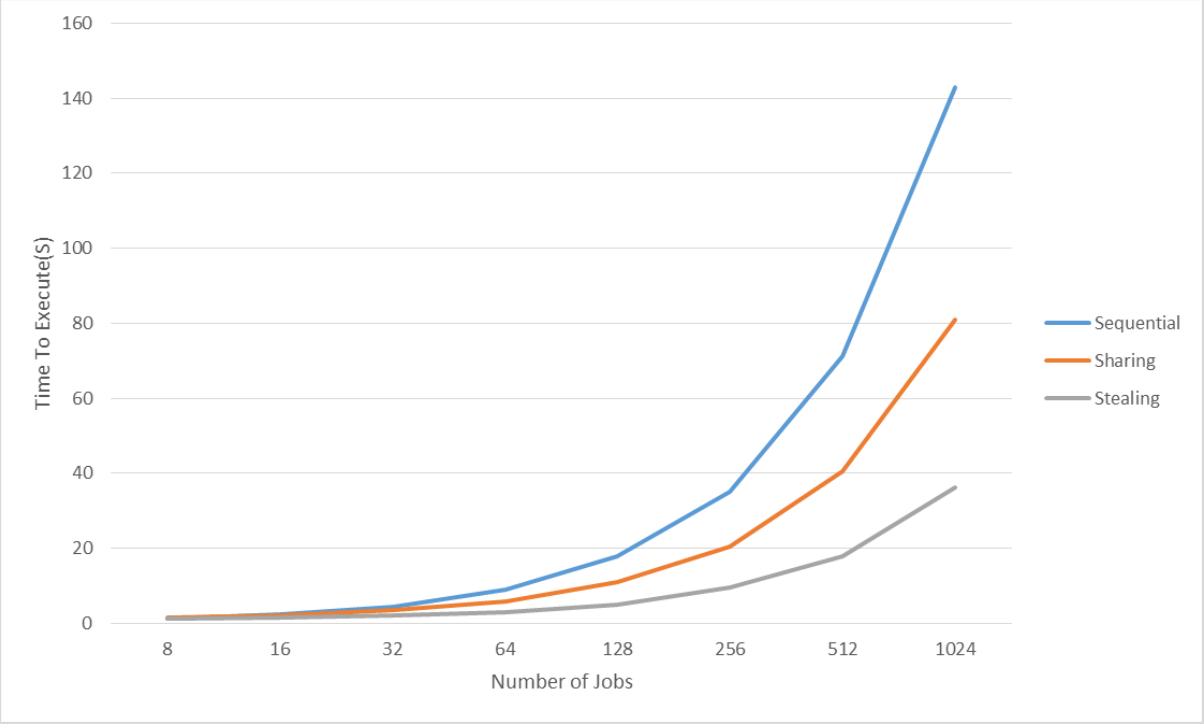


Figure 33 - Lifetime vs Number of Jobs

The first thing that is noticeable about this graph, is that as the number of jobs increases the time that the sequential execution takes to complete these jobs grows at an almost exponential rate. As well as this, it should also be noted that for the first two job sets (8 & 16) there is little difference between the parallelised and sequential implementation. This is due to the start-up cost associated with the framework negating any performance gains under such small loads.

What we can also see from this graph is that the work stealing algorithm, as expected, has superior performance as the amount of work available increases. The reasoning for this, is that as time goes on the work stealing algorithm spends a progressively greater amount of its lifetime directly executing work (Figure 34). This is compounded by the fact that as the number of jobs within the system increases, the number of successful attempts to steal work increases (Figure 35).

Despite this, it is also worth noting that for both of these algorithms we do not see an increase in performance equivalent to the additional number of threads within the parallel algorithms when compared to sequential execution. This is due to the cost associated with the creation and deletion of these threads by the operating system. However as the job count continues to grow, and the amount of time taken to complete work increases the effect of this will be lessened.

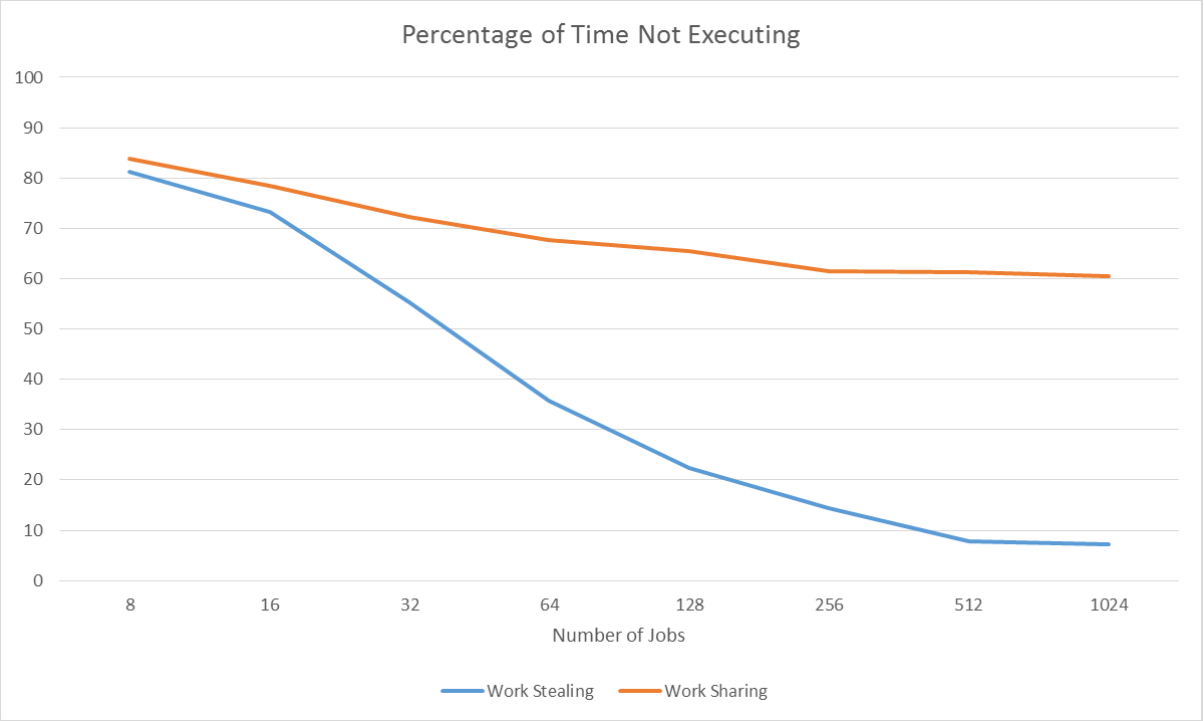


Figure 34 - Percentage time not directly executing Tasks for varied number of jobs

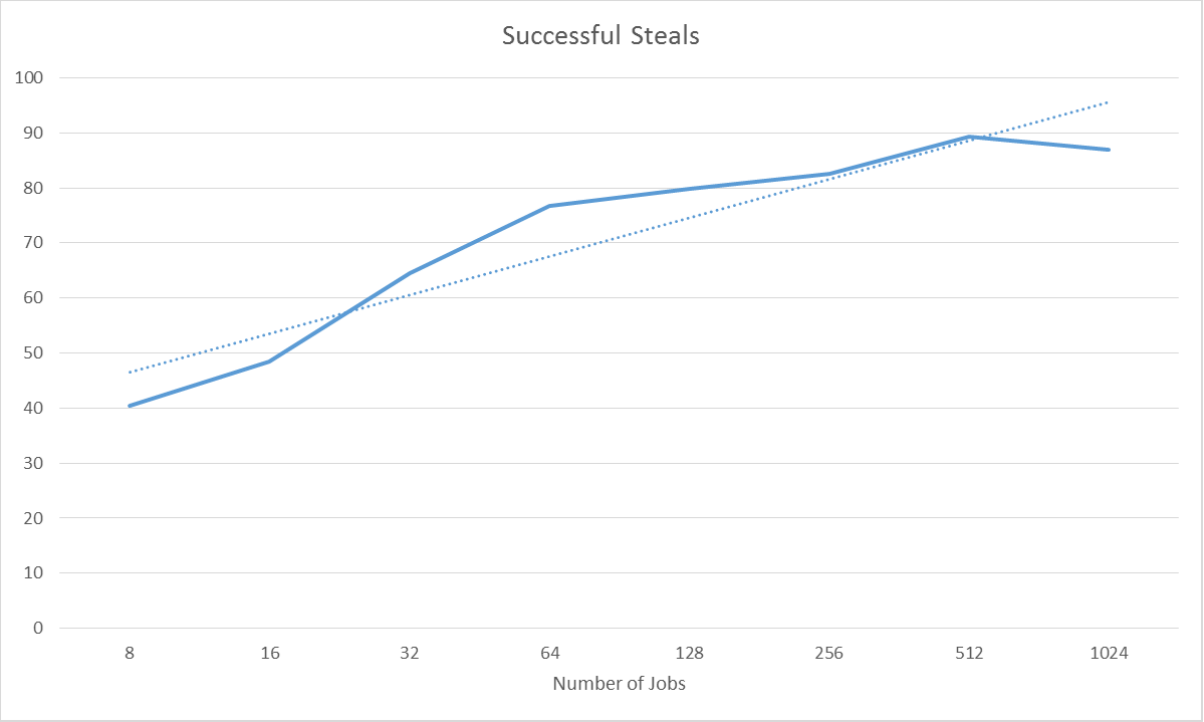


Figure 35 - Percentage of successful steal attempts for varied Job count

6.3.3 Varied Input Grid Size

This test was focused on determining how the performance characteristics of each algorithm changes, as the computational complexity of the input increases. To perform this test a constant number of jobs were created with an *N x N* input, where the *N* value increases by 10 at each iteration of the test. This test was initially performed with 8 jobs, however after seeing that for 8 inputs all three algorithms were relatively equivalent (Figure 33) it was decided to increase the count to 128.

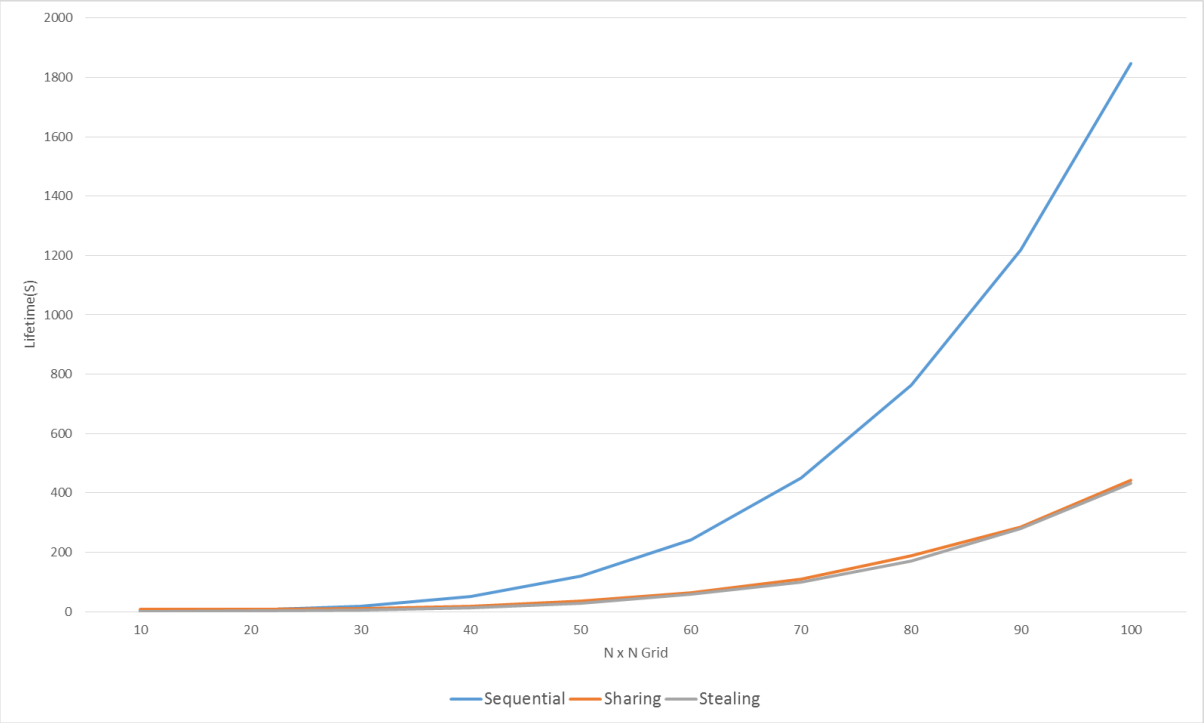


Figure 36 - Execution time of algorithms under input of 128 N x N *Grids*

As can be seen from the above graph, after the 30 x 30 grid the time taken to execute sequentially grows extremely rapidly to the point that executing the 100 x 100 Grids requires more than half an hour to complete. By comparison both work sharing and stealing show very similar growth characteristics, which are approximately 4 times faster than the non-parallelised execution.

The reason that stealing is not superior in this repspect, is owing to the fact that as the size of the Grid increased the percentage of successful steal attempts decreases (Figure 37). This degredation of the stealing mechanism therefore means that there is a higher likelihood of worker threads experiencing starvation, which subsequently negates any performance increase that the stealing algorithm has compared to work sharing. It is likely that the worker threads that experienced starvation in this way were unsuccessul due to one or more worker threads executing the longer running Tasks after the other threads have completed all of the outstanding Tasks. This means that no new Tasks would be available until the longer running Task had completed, which could lead to a spike in the number of steals and a drop in their success rate.

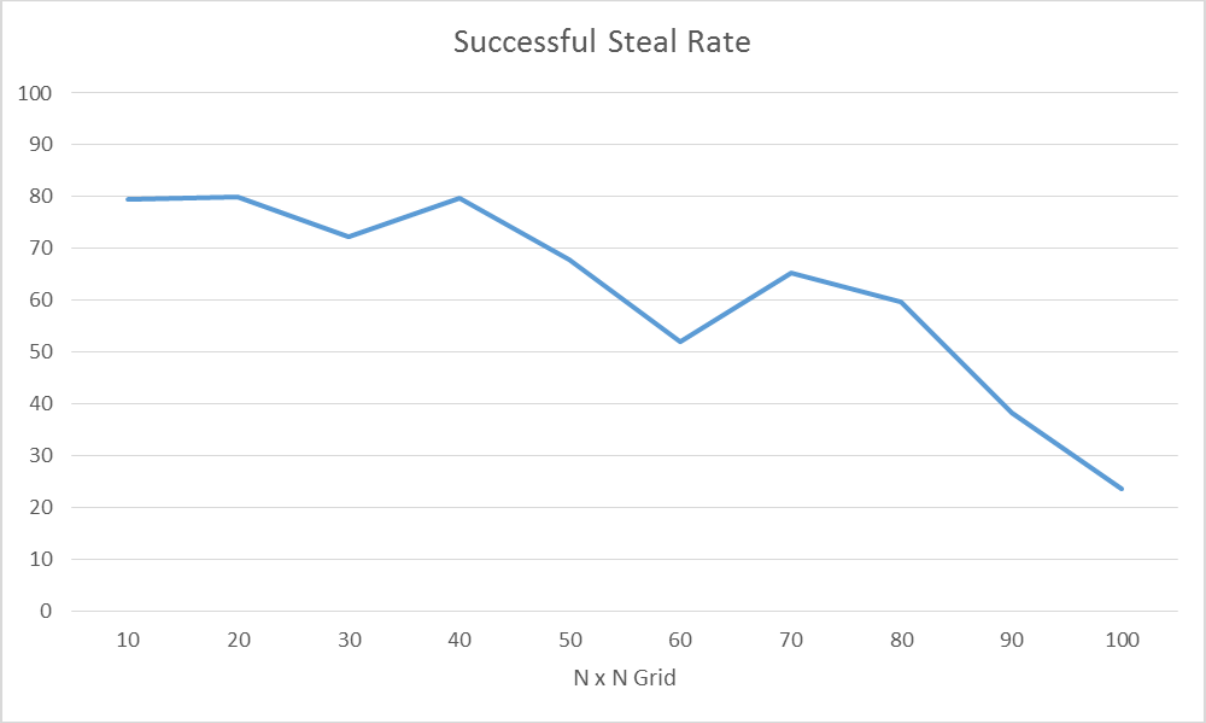


Figure 37 - Successful Steal Rate for variety of grid sizes

6.4 Profiling

Instrumentation based profiling was utilised on this project to obtain CPU and memory based performance characteristics for both of the scheduling algorithms. This was necessary due to the fact that although the measurements utilised for the experimentation phase of this project were useful in many ways, they provide a much less accurate picture of the system performance than can be obtained by the profiler.

Within the profiler several flags were set such that cache misses, non-halted cycles and instructions retired could be measured. The first of these measurements indicates the number of times the CPU had to access the next level of cache memory due to the relevant data not being present in the current cache, which can have an effect on performance if too high. The second of these measures the number of CPU cycles that were not stopped during execution. The final measurement, indicates the number of instructions executed by the CPU over the lifetime of the system.

Figure 38 - CPU utilisation under Work Stealing

The above graph, depicts the CPU utilisation of the system when Work Stealing was employed to process 5000 jobs. All of the graphs obtained under Work Stealing show a similar trend in that they have extremely high utilisation for their duration. This is in contrast to the graphs produced under work sharing, pictured below, which have a more trough like appearance. The reasoning for this, is that work sharing will perform Tasks in the order in which they are added into the scheduler. For example, if given 5 jobs it would firstly load in all of the grids, then perform the pathfinding and finally the output would be produced for all. The Work Stealing method contrasts this as the Tasks are performed in a seemingly random order, as the stealing process takes Tasks from the end of other threads deques thereby meaning that one thread could be producing an output, whilst another performs pathfinding whilst yet another is reading from the input file. This therefore leads to a more uniform utilisation of the CPU.

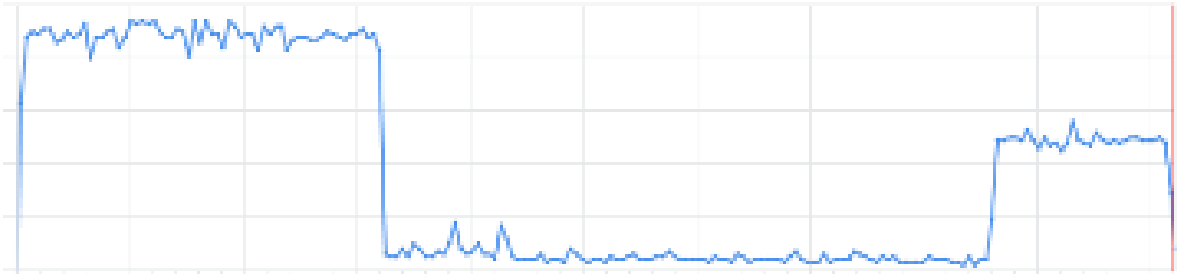


Figure 39 - CPU utilisation under Work Sharing

From the CPU measurements mentioned previously, we are able to produce two further measures. The first is the percentage of instructions that experience cache miss, this allows us to see to the frequency that cache misses occur within the system. The second, is instructions per cycle, which tells us how many instructions are fulfilled within each clock cycle. A value greater than 1 for this measure indicates a good utilisation of the CPU.

	Instructions Retired	Cache Misses	Non-Halted Cycles	% Cache Misses	Instructions Per Cycle
0	528052490	337436	579642673	0.063901981	0.910996575
1	356489052	215784	332159249	0.06053033	1.073247405
2	1456307488	680307	1426158517	0.046714516	1.021139986
Ave	780283010	411175.6667	779320146.3	0.052695709	1.001235517

Table 2 - Profiling data for executing 5000 jobs under Work Stealing

These measurements were collected for three job sets (500, 1000 & 5000) from which the following graphs were produced.

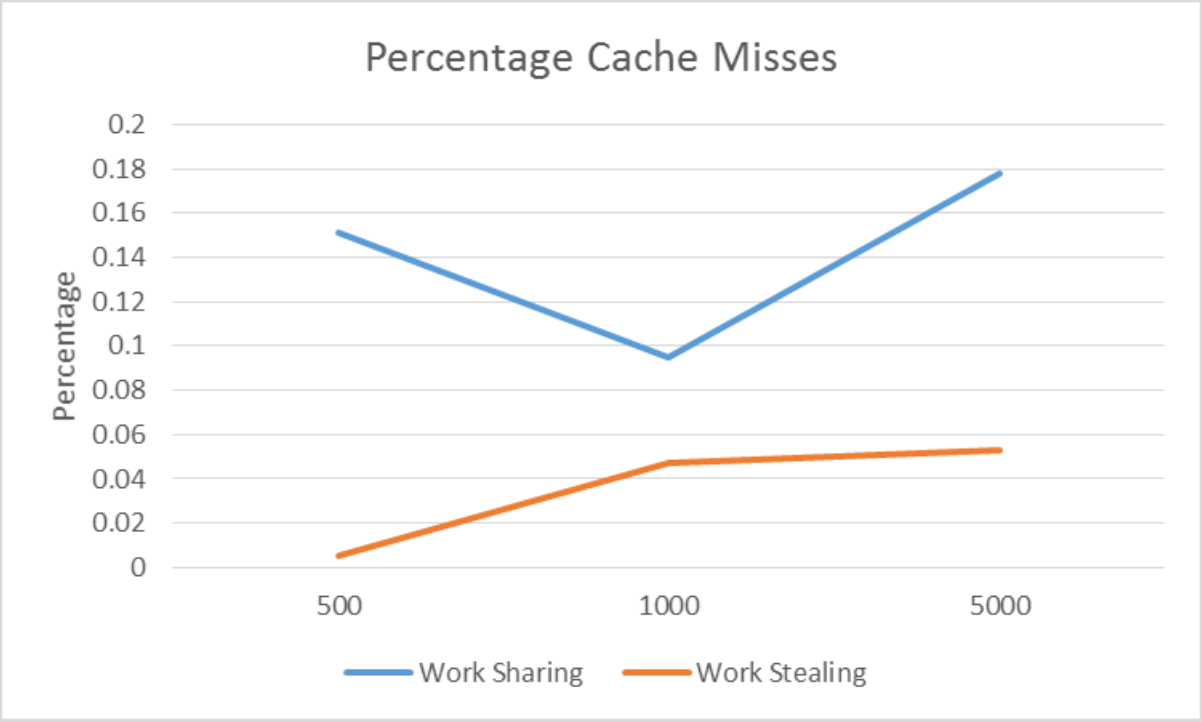


Figure 40 - Percentage cache misses for both algorithms

From the above graph, we can see that regardless of the algorithm and job count the percentage of cache misses that occur are far less than 1% of all instructions performed. This therefore indicates that the system as a whole doesn't experience many cache misses, which is good in terms of performance as only a small amount of time is spent accessing lower level memory which can have an effect on performance.

What we can gather from Figure 41, is that on the whole the system has good utilisation of the CPU as the number of instructions executed per CPU cycle is at its lowest 0.96. As well as this, we can see that as the number of jobs increases we see that the Work Stealing algorithm shows a slight decrease in the number of instructions per cycle. We can attribute this characteristic to the fact that as the number of jobs increases, the number of steal attempts also increase. Each of these steal attempts increases the number of instructions executed, but equally increases the number of lock acquisitions required. Each lock acquisition within

.Net requires approximately 50ns (Mischel, 2003), which although small for one acquisition can cause long delays if the number is high.

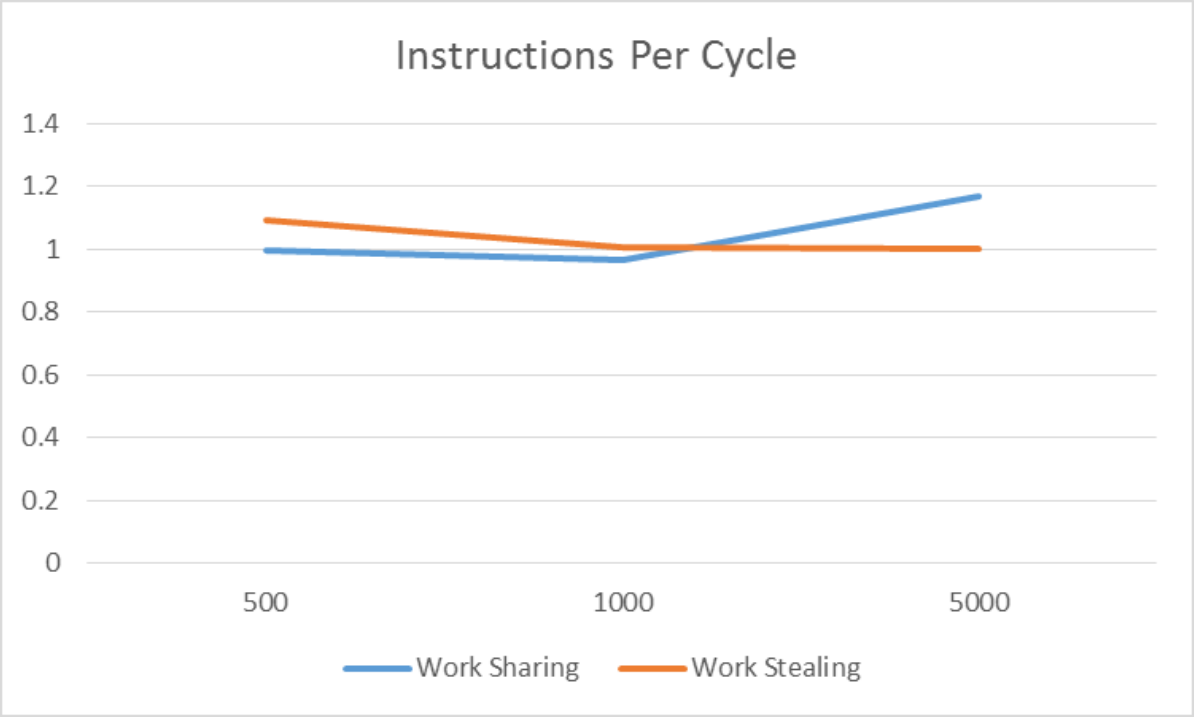


Figure 41 - Instructions per cycle for both algorithms

7 Evaluation

7.1 Project Achievements

Owing to the fact that all four primary objectives have been fulfilled within the project, as well as the secondary objective, it can be considered a successful project. The following section will detail how each objective was fulfilled.

Objective 1 – Develop a classification mechanism that enables users to define areas of their software as Tasks.

This project enables users to create an instance of the Task object, which when instantiated encapsulates any code that is passed into the constructor through the delegate that is the classes only parameter. By utilising a delegate, it is possible for the user to pass either a function pointer (i.e. the name of the function that they wish to execute in parallel) or an anonymous function into the Task object which provides more flexibility to the user as they can implement parallelism regardless of how they have written their software.

```
Task task = new Task(() =>
	{
		for (int i = 0; i < 10; ++i)
		{
			Console.WriteLine(i.ToString());
		}
	}
);
```

Figure 42 - Passing an anonymous function into the Task class

Objective 2 – Implement a scheduling algorithm to allow multiple tasks to run simultaneously.

The framework that has been developed for this project has the ability to utilise two distinct scheduling algorithms, both of which utilise the thread pooling mechanism. The first of these algorithms, Work Sharing, functions by assigning work when a worker thread becomes underutilised. As can be gathered from the experimentation section of this report, although the algorithm is not as effective at utilising the CPU as the second algorithm that was implemented it still has the ability to execute multiple tasks in parallel in a manner that utilises the CPU far more effectively than a serial implementation could achieve.

Objective 3 – Design and implement a framework that can be easily implemented within a software project by a user.

The system that has been implemented has a very simplistic interface that can be accessed by the user. The user themselves can only access the Task object and a very small number of functions within the scheduler. The rest of the functionality of the system is inaccessible to them due to the access modifiers associated with the different components.

Given that the user can only access such a small subset of the functionality of the system but is still able to create tasks and execute them in parallel, the framework itself is easy to use through its sheer simplicity.

Objective 4 – Optimise the framework so that it can maximise its utilisation of the CPU's processing capability.

Given the data obtained from both the profiler and the various experiments performed it would appear that both algorithms utilise a very high percentage of the CPU's processing power when they are executing. As such, the system can be said to efficiently utilise as much of the CPU's capability as is available.

Objective 5 – Produce a framework that implements more than one scheduling algorithm.

As has been mentioned, the system implements two distinct scheduling algorithms. The second of these is work stealing, which functions by putting the onus for obtaining work into the hands of the worker threads rather than being reliant upon the scheduler itself. By implementing this secondary system, it was possible to perform a variety of different experiments and determine which of the scheduling mechanisms was superior under a variety of circumstances.

7.2 Future Work

Although the project has successfully achieved all of the objectives it originally set out to, there are a few places where the system could be improved in the future.

7.2.1 Use of Concurrent Collections

Within .NET 4.0, there are data structures that are available which provide built in thread safety through the namespace System.Collections.Concurrent. These data structures could be used relatively extensively throughout the system, to provide thread safe access to data sets without the need to obtain and release a lock for the entire class instance that is encapsulating the data. By doing this it will be possible to significantly reduce the number of lock contentions that occur within the system, which can be a limiting factor to performance if too high.

7.2.2 Expand Task and associated delegate

One of the shortcomings of this project, is that at present it is not possible to produce Tasks that take parameters or return values. Neither of these would be a simple fix, due to the fact that parameters for methods can have infinite permutations and that delegates do not return values when they are invoked. However as at present only methods which return void and have no parameters can be used to create Tasks, making these improvements would significantly alter the user experience for the better.

7.2.3 Fault Tolerance

As was mentioned previously, the method in which the system currently handles exceptions is both naïve and not particularly elegant. To resolve this, it would be necessary to implement some form of fault tolerance into the system. Fault tolerance, is a method through which a system can recover in the event of a fault or failure occurring within the system.

To implement this within the system, it would be necessary to allow the user to specify how they wished the system to proceed if a Task that they have created was to cause an exception to occur within the system. For example, they could choose to remove the Task and its dependencies if it has any or the user could choose to rerun the Task in the hope that the exception only occurred on one occasion.

7.2.4 Subtasking with Work Stealing

At present if the user utilises subtasking to create interdependent Tasks whilst also using the work stealing algorithm, the likelihood of all Tasks completing is very small due to the nature of the algorithm. As such, it would be beneficial to implement a mechanism that ensures that the system does not close prematurely under these circumstances.

To do this it would be necessary to have a property within the scheduler that blocks the closing mechanism until all of the worker threads have completed all of the Tasks in their deque. This property could be integrated into the algorithm selection mechanism already present within the scheduler, which would allow the framework to retain its simplicity.

8 Conclusion

This project has resulted in the development of a system employing two distinct scheduling algorithms which are both capable of enabling users to execute defined work in parallel. These two algorithms have subsequently been tested, experimented upon and profiled to the point that under a variety of circumstances it is now possible to state which algorithm would be best suited for a specific problem.

As with all software development projects, there is always room for improvement and this project is no exception. Although the system as a whole functions as expected, there are a number of modifications that could be made which could potentially improve the usability and the performance characteristics of the system.

The overall aim of this project was to design and implement a framework that allows users to easily utilise task based parallelism within their project, which in turn would allow users to capitalise on the greater processing power available to software developers due to the advent of ‘massively multi-core’ CPU’s. Given this aim and the fact that all of the objectives that were detailed at the start of this project have been realised, this project could be said to have been a resounding success.

Appendix A: Pathfinding Background

Pathfinding

There are a variety of algorithms that can be used to implement path finding within a computer application, with a large number of them being based upon the work that Edsger Dijkstra detailed in his 1959 paper (Dijkstra, 1959) on finding the shortest path on a weighted graph. To be able to understand the principles detailed by Dijkstra it is first necessary to briefly outline what a weighted graph is.

Weighted Graph

A graph, in its simplest form, is comprised of a set of nodes and a set of edges. The nodes can be seen to represent specific points within the graph, whilst the edges are connections between two of these nodes. For example, Figure 8 below contains A & B, which represent two nodes, and E which represents the edge connecting them both.

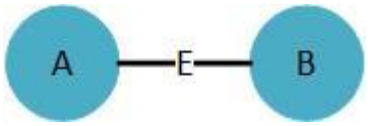


Figure 43 - Basic Graph

The weighted graph is distinct from a standard graph, as it introduces a value representing weight to each of the edges. These weights can be representative of distance, cost, time or any number of other factors depending on what the graph is representing. If Figure 8 was a weighted graph, E would also have a weight associated with it that represents the value required to traverse between the two nodes.

Djikstra's Algorithm

The algorithm described by Dijkstra functions, firstly by assigning every node a tentative distance: 0 for the starting node and infinity, or the maximum value of an integer, for all other nodes. Following from this, the starting node will be assigned as the current node and all others will be marked as unvisited, this is usually done by creating a visited and unvisited set and assigning all nodes that are not the starting node into the unvisited set.

From here, we need to visit all of the current nodes 'neighbours', nodes that are connected to the current node through an edge, and calculate their 'tentative distance', which is the value of current node plus the weight of the edge connecting the two nodes. For example if node A has a value of 10 and is connected to node B by an edge with value 12, then node B's tentative distance would be that of 22. Once this value has been calculated we then compare this value to the current value of the neighbouring node and assign the smaller of the two values as the value of the node.

After this we mark the current node as visited, by assigning it to the visited set. If the destination node has been visited at this point then the algorithm will terminate, however if this is not the case then we select the node with the smallest value that is unvisited and assign that as the current node which will then go on to visit its neighbours and repeat the selection process.

The original algorithm that was detailed had a quadratic time complexity, specifically $O(|V|^2)$ where $|V|$ is representative of the number of distinct nodes. This time complexity can be improved through the use of a min-priority queue (Fredman & Tarjan, 1987), a data structure that stores the data with an associated priority value (in this case the nodes distance value) and returns the data point with the lowest priority upon request, which reduces the need to search the unvisited set at every iteration and subsequently creates an algorithm with a

complexity of O(|E| + |V| log |V|), where |E| is the number of edges associated with the number of graphs.

Best First Search

This algorithm works in a similar manner to Dijkstra's however it differs due to the fact that when it selects the next node for the path it utilises the one that is closest to the goal node, in lieu of using the node that is closest to the current node.

All Best First Search based algorithms, have a worst case time complexity of $O(b^d)$, where *b* is the branching factor and d represents the depth from the start that the goal resides.

Greedy Best First Search

The Greedy Best First Search (GBFS), is a variant on the basic Best First Search in that it is a greedy algorithm (Black, 2005), which means that it takes the best local solution while finding the answer as opposed to finding the best global solution. What this means is that when the GBFS algorithm determines the next node to visit, this will be added to the route to be taken and will not be reconsidered at a later point as is the case with Dijkstra's. This therefore can lead to less than optimal solutions for some path finding problems.

A*

A* is another variation of the Best First Search algorithm, this algorithm differs from the standard algorithm in the heuristic that it utilises. A* uses a combination of the approach used in Dijkstras algorithm, namely the node that is closest to the current node, and that of Best First Search, selecting the node closest to the goal. These two values are summed together, through the formula $f(n) = g(n) + h(n)$ where $g(n)$ is the edge cost and $h(n)$ is the heuristic cost, and the next node that is selected is the node with the lowest summed value ($f(n)$). Through the use of this heuristic this algorithm provides the accuracy of Dijkstra's algorithm with the benefit that the number of nodes visited, and therefore time taken, is reduced.

Grids

Although pathfinding algorithms are designed for weighted graphs, it is also possible to implement them using a grid. The blocks within a grid could be used to represent a node, and the edges associated with a node could be determined by what other blocks are within a certain radius of the centre block. In the example below blocks *A* through *I* would represent nodes. If we used a radius of 1 block, then *A* would have three edges (*B,D,E*) whilst E would have eight (*A,B,C,D,F,G,H,I*).

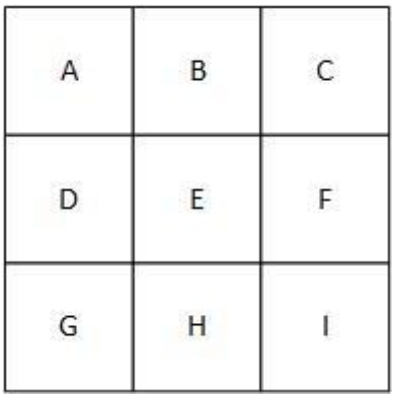


Figure 44 - Example of Grid

Appendix B: Pathfinding Algorithm Implementation

```
public void Dijkstra()
{
	PriorityQueue<Square> frontier = new PriorityQueue<Square>();
	Dictionary<Square, Square> comeFrom = new Dictionary<Square, Square>();
	Dictionary<Square, int> costSoFar = new Dictionary<Square, int>();
	List<Square> neighbours = null;
	Square current = null;
	int newCost = 0;

	frontier.Enqueue(_start, 0);
	comeFrom[_start] = _start;
	costSoFar[_start] = 0;

	while (frontier.Count() != 0)
	{
		current = frontier.Dequeue();
		if (current == _end)
		{
			break;
		}
		neighbours = current.GetNeighbours();
		foreach (Square b in neighbours)
		{
			newCost = costSoFar[current] + Cost(b);
			if (!costSoFar.ContainsKey(b) || newCost < costSoFar[b])
			{
				costSoFar[b] = newCost;
				frontier.Enqueue(b, newCost);
				comeFrom[b] = current;
			}
		}
	}
	_djikstraRoute = Route(comeFrom);
}
```

Figure 45 - Dijkstra's Algorithm

```
public void AStar()
{
	PriorityQueue<Square> frontier = new PriorityQueue<Square>();
	Dictionary<Square, Square> comeFrom = new Dictionary<Square, Square>();
	Dictionary<Square, int> costSoFar = new Dictionary<Square, int>();
	List<Square> neighbours = null;
	Square current = null;
	int newCost = 0;
	int priority = 0;

	frontier.Enqueue(_start, 0);
	comeFrom[_start] = _start;
	costSoFar[_start] = 0;

	while (frontier.Count() != 0)
	{
		current = frontier.Dequeue();
		if (current == _end)
		{
			break;
		}
		neighbours = current.GetNeighbours();
		foreach (Square sq in neighbours)
		{
			newCost = costSoFar[current] + Cost(sq);
			if (!costSoFar.ContainsKey(sq) || newCost < costSoFar[sq])
			{
				costSoFar[sq] = newCost;
				priority = newCost + Heuristic(_end, sq);
				frontier.Enqueue(sq, priority);
				comeFrom[sq] = current;
			}
		}
	}
	_aStarRoute = Route(comeFrom);
}
```

Figure 46 - A Algorithm*

References

Black, P. E., 2005. *Greedy Algorithm.* [Online]
Available at: http://www.nist.gov/dads/HTML/greedyalgo.html
[Accessed 8 May 2016].

Blumofe, R. D., 1995. [Online]
Available at: http://supertech.csail.mit.edu/papers/rdb-phdthesis.pdf
[Accessed 15 January 2016].

Boost Software, 2015. *Boost C++ Libraries.* [Online]
Available at: http://www.boost.org/
[Accessed 17 January 2016].

Dijkstra, E., 1959. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik,* Issue 1, pp. 269 - 271.

Dijkstra, E. W., 1965. Solution of a Problem in. *Communications of the ACM,* 8(9), p. 569.

DotGNU Project, 2012. *DotGNU Project.* [Online]
Available at: http://dotgnu.org/
[Accessed 11 November 2015].

Fredman, M. L. & Tarjan, R. E., 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM,* 34(3), pp. 596 - 615.

Gamma, E. H. R. J. R. V. J., 1995. Singleton. In: *Design Patterns.* s.l.:s.n., p. 144.

Gross, T., O'Hallaron, D. & Subholk, J., 1994. Task Parallelism in a High Performance Fortran Framework. *IEEE Parallel & Distributed Technology,* Issue Fall 1994, pp. 16-25.

Hillis, W. & Steele Jr, G., 1986. Data Parallel Algorithms. *Communications of the ACM,* 29(12), pp. 1170-1183.

IEEE Spectrum, 2015. *The 2015 Top Ten Programming Languages.* [Online]
Available at: http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages
[Accessed 11 November 2015].

Microsoft, 2015. *Common Language Runtime.* [Online]
Available at: https://msdn.microsoft.com/en-us/library/8bs2ecf4
[Accessed 11 November 2015].

Microsoft, 2015. *Lambda Expressions (C# Programming Guide).* [Online]
Available at: https://msdn.microsoft.com/en-gb/library/bb397687.aspx
[Accessed 8 April 2016].

Microsoft, 2015. *Mutex Class.* [Online]
Available at: https://msdn.microsoft.com/en-us/library/system.threading.mutex(v=vs.110).aspx
[Accessed 21 November 2015].

Microsoft, 2015. *Overview of Syncronization Primitives.* [Online]
Available at: https://msdn.microsoft.com/en-us/library/ms228964(v=vs.110).aspx
[Accessed 21 November 2015].

Microsoft, 2016. *How to: Use a Background Worker.* [Online]
Available at: https://msdn.microsoft.com/en-us/library/cc221403(v=vs.95).aspx
[Accessed 11 January 2016].

Microsoft, 2016. *lock Statement.* [Online]
Available at: https://msdn.microsoft.com/en-us/library/c5kehkcz(v=vs.110).aspx
[Accessed 12 April 2016].

Microsoft, 2016. *Thread Class.* [Online]
Available at: https://msdn.microsoft.com/en-us/library/system.threading.thread(v=vs.110).aspx
[Accessed 11 January 2016].

Mischel, J., 2003. *The Cost of Locks.* [Online]
Available at: http://www.informit.com/guides/content.aspx?g=dotnet&seqNum=600
[Accessed 5 May 2016].

Moore, G. E., 1965. Cramming more components onto integrated circuits. *Electronics,* 19 April, 38(8), pp. 114-117.

Oxford Dictionaries, Oxford University Press, n.d. *Concurrent.* [Online]
Available at: http://www.oxforddictionaries.com/definition/english/concurrent
[Accessed 3 April 2016].

Oxford Dictionaries. Oxford University Press, n.d. *Parallelism.* [Online]
Available at: http://www.oxforddictionaries.com/definition/english/parallelism
[Accessed 12 October 2015].

Robert D. Blumofe, C. E. L., 1999. Scheduling Multithreaded Computations. *Jurnal of the Association for Computing Machinery,* 46(5), pp. 720 - 748.

Robinson, A., 2007. *CodeGuru.* [Online]
Available at: http://www.codeguru.com/cpp/sample_chapter/article.php/c13533/Why-Too-Many-Threads-Hurts-Performance-and-What-to-do-About-It.htm
[Accessed 12 October 2015].

Skeet, J., 2013. Delegates. In: *C# in Depth - Third Edition.* s.l.:Manning Publications, pp. 30 - 38.

Stallings, W., 2011. In: *Operating Systems: Internals and Design Principles.* 7th ed. s.l.:Prentice Hall, p. 220.

Thomas H. Cormen, C. E. L. C. S. R. R., 2009. In: *Introduction to Algorithms.* s.l.:M.I.T Press, p. 773.

University of Illinois, 2008. *Parallel @ Illinois.* [Online]
Available at: http://graphics.cs.illinois.edu/sites/default/files/upcrc-wp.pdf
[Accessed 5 October 2015].

Zimmerman, J., 2013. *Principles of Imperative Computing - Unit Testing.* [Online]
Available at: http://www.cs.cmu.edu/~rjsimmon/15122-s13/rec/07.pdf
[Accessed 08 April 2016].