Dorian Peake

Parallelism in OCaml Under the JVM

Computer Science Tripos

St John's College

May 16, 2014

Proforma

Name:	**Dorian Peake**
College:	**St John's College**
Project Title:	**Parallelism under OCaml using the JVM**
Examination:	**Computer Science Tripos, July 2014**
Word Count:	**TBC**
Project Originator:	Dorian Peake
Supervisor:	Jeremy Yallop

Original Aims of the Project

The aim of this project was to highlight the differences between single and multithreaded OCaml code. This was done by adapting the Lightweight Threading Library (LWT) – a very popular threading library for OCaml – to use the OCaml to Java bytecode compiler OCaml-Java, thus allowing software written with LWT to utilise the multithreading capabilities of the JVM.

Work Completed

The LWT threading capabilities have been integrated with OCaml-Java via the development of my interface, thus allowing a subset of LWT programs to compile to Java bytecode and run in parallel on the JVM. This is done by scheduling lightweight threads between Java worker threads which continuously execute code until interrupted. Test code written for this interface achieve similar speedup gains to code written directly in Java (speedup of 3.5 – 4 with the interface compared to a speedup of 5 – 5.5 on normal Java), with an again similar speedup curve – which is a very positive result.

Special Difficulties

None

Declaration

I, Dorian Peake of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

List of Figures

Acknowledgements

This project was a success thanks to the help of my supervisor Jeremy Yallop, Leo White at the OCaml Labs, Malte Schwarzkopf for his motivation and gift ideas and finally Xavier Clerc for his example code which made developing the interface a whole lot easier.

Chapter 1

Introduction

1.1 Motivation

Multi-core processors are the focus of chip development in this current day and age. Systems need to continue to increase chip parallelism as this has become the main approach for achieving faster code execution since CPU frequency speeds plateaued around 2003[12]. Of course it is also important for languages and compilers to utilise these advancements in technology, otherwise they will soon become useless for modern software development and developers will begin to seek more appropriate alternatives that provide them the development resources they need. There also exist software libraries such as OpenCL that are specifically designed to utilise extra computational power from your GPU which is normally inaccessible for general processing tasks. Clearly parallel processing is a very important topic and is currently a very active research area therefore it is well worth discussion.

Functional Programming concepts have been around since Turing and Church's work in the 1930's[16] but have become more popular in recent times. The expressiveness that functional programming paradigms give the programmer has finally caught on in mainstream development and therefore languages such as C++ and Java are incorporating notable functional programming techniques such as type inference[6] and first-class anonymous functions (lambdas)[8]. Notable functional languages include Haskell and ML (which has inspired more widely used functional languages for example OCaml[14]). OCaml is a very extensive functional language which also includes an elegant object system, which incorporates the best of both functional and imperative programming techniques. The ability to develop in these two different programming styles

interchangeably makes OCaml a very powerful language to develop with. Unfortunately OCaml in it's current state is unable to compile parallel code, which of course is becoming an ever more necessary approach to development as systems become more and more parallel.

Java is a mature imperative programming language/system widely used in industry to build multithreaded software[15]. It compiles to Java bytecode which runs on it's virtual machine which in turn is able to run on everything from ATM's and credit card chips to home desktops and servers – leading to one of the Java slogans '*write once, run everywhere*'. Over the years the JVM has become highly optimised to the hardware/systems that it runs on such that there is little difference between Java code running on the JVM and natively compiled code. Furthermore, Java supports parallel processing, fully utilising any number of cores/hardware threads that the system has to offer.

A project called OCaml-Java aims to merge the best aspects of both OCaml and Java together to create a better OCaml programming solution. It achieves this by compiling OCaml code to Java bytecode, thus allowing it to run on almost any JVM. This means two things:

- OCaml can now be written to run on a wide variety of hardware/systems.
- OCaml can use Java's multithreading capabilities to run fully parallel OCaml code[4].

OCaml-Java comes with a basic collection of concurrency tools which are essentially a small subset of Java's own concurrency library with wrappers that allow access from within OCaml. Unfortunately the concurrency techniques provided by Java do not nicely translate to the way concurrency is done in a functional programming language like OCaml. There do exist commercially-used, optimised OCaml concurrency libraries such as Async by Jane Street and LWT (Lightweight Threads Library) by Ocsigen, however these do not function when compiled to Java bytecode via OCaml-Java as they rely on the Unix bindings and threads provided either by the OCaml library or similar ones constructed in house. If we could get one of these OCaml concurrency libraries to work when compiled with OCaml-Java, we could demonstrate OCaml-Java's true potential to create parallel OCaml code, and OCaml's true capabilities as a parallel programming language – this is the motivation for my project.

My project aims to interface OCaml-Java and the LWT concurrency library in order to demonstrate OCaml's potential as a parallel programming

language. The rest of this document outlines the preparation and research I conducted in order to plan my approach to this project (chapter 2); the methods and processes that were used to implement the project (chapter 3) and the result of the project development (chapter 4) and the work process as a whole (chapter 5). My code can also be found in appendix B.

Chapter 2

Preparation

This chapter outlines the processes taken in preparation for the project's implementation. I begin with a look into OCaml's multithreaded capabilities, including the limitations around developing parallel OCaml code and why these limitations exist. I also compare LWT's own asynchronous library in comparison to Java's and the difficulties that may occur when attempting to interface the two. Finally there is an overview of OCaml-Java, including what tools become available with its use, and its concurrency modules which provide the user with the ability to write parallel OCaml code.

2.1 Investigating Multithreaded OCaml

In order to outline the best implementation plan for the LWT–OCaml-Java interface, it was necessary to understand OCaml's limitations in compiling parallel code. The OCaml-Java paper from the 2012 'Trends in Functional Programming' collection[4] states the reason – the runtime OCaml library is not reentrant and the OCaml garbage collector is neither parallel nor concurrent. Making the runtime library reentrant is a relatively straightforward change to the system in comparison to making the garbage collector parallel – which is far more involved – therefore the real problem is developing a concurrent and parallel garbage collector. Java's G1 garbage collector is fully parallel[7] and as a result provides multithreaded support of Java applications. Therefore by compiling OCaml code to Java bytecode, the OCaml language may utilise Java's parallelism capabilities.

It was necessary to decide which of the popular concurrency libraries available to OCaml I should use for the purposes of this project therefore I researched

into OCaml's ability to produce concurrent OCaml code. This research led me to OCaml's own inbuilt threads library. These native threads are designed for single core asynchronous development as the OCaml manual[13] describes OCaml threads as executing concurrently on a single processor using time sharing. There exist other concurrency libraries which boast a richer feature set than the one provided by OCaml as default. Examples of these libraries include Async – developed by Jane Street Capital[1] – and LWT (Lightweight Threads Library) – developed by Ocsigen[2]. These are the two main concurrency packages used within OCaml development, choosing one to use within my project boiled down to which one was most popular. My reasoning is that they are both extremely good packages and would serve well to demonstrate the capabilities of multithreaded OCaml-Java. LWT won this battle.

2.2 Investigating LWT

LWT is an extensive concurrency suite for OCaml which is based around the idea of lightweight cooperative threads[11] where cooperative threads manage their own life cycle as opposed to preemptive threads which are controlled externally, e.g. by the operating system. Creating and destroying a thread in LWT are such cheap operations that the system will create a thread for every system call that is executed. This idea of lightweight threads with a small lifetime is well suited to the way that OCaml (and languages within the ML family) work which is by allocating new immutable data frequently which tend to have very small lifetimes. The OCaml garbage collector is also optimised for this by using generational garbage collection techniques to more frequently collect younger data since data that has survived a garbage collection once or more before is likely to have an extended lifetime[2].

On the other hand, Java threads have a greater thread creation overhead and as such cannot be used similarly to the way LWT threads are used without detrimental effects to performance. Java's `java.util.concurrent` package avoid the thread creation overheads by utilising thread pools instead[10]. This observation led me to decide that a good implementation strategy for the interface would be to create a thread scheduler that maps LWT threads to Java threads as a 1:1 mapping would simply expose great amounts of thread creation overhead, possibly crippling any speedup gains parallelism would obtain.

[1] Async Library – https://github.com/janestreet/async
[2] Lightweight Threads Library – https://github.com/ocsigen/lwt

There are many functions within LWT to create threads. One of the most commonly used is the `bind` function, with the following type:

```
val bind : 'a t -> ('a -> 'b t) -> 'b t
```

The bind function constructs a thread from a thread `t` and a function which takes the return value of `t`. After performing some processing, bind returns a new thread. What this processing does is attach the function as '*waiter*' on the thread. A waiters purpose is to wait until the thread has finished executing, then perform its own execution with the return value of the thread. This serialises the execution of two functions in an asynchronous environment and acts synonymously to the classic 'sync' operation within various writings on parallel processing[5]. The thread returned by the bind function will contain the final result of the waiter function once everything has finished executing. Until then, the function thread will remain in the `Sleep` state (moving to the `Return 'a` state once execution has completed). The following listing is an extract from the LWT code base showing the possible states a thread may exist in:

Listing 2.1: Thread state type

```
type 'a thread_state =
  | Return of 'a
      (* [Return v] a terminated thread which has successfully
         terminated with the value [v] *)
  | Fail of exn
      (* [Fail exn] a terminated thread which has failed with the
         exception [exn]. *)
  | Sleep of 'a sleeper
      (* [Sleep sleeper] is a sleeping thread *)
  | Repr of 'a thread_repr
      (* [Repr t] a thread which behaves the same as [t] *)

and 'a thread_repr = {
  mutable state : 'a thread_state;
  (* The state of the thread *)
}

and 'a sleeper = {
  mutable cancel : cancel;
  (* How to cancel this thread. *)
  mutable waiters : 'a waiter_set;
```

```
22    (* All thunk functions. These functions are always called
       inside a
23       enter_wakeup/leave_wakeup block. *)
24    mutable removed : int;
25    (* Number of waiter that have been disabled. When this number
26       reaches [max_removed], they are effectively removed from
27       [waiters]. *)
28    mutable cancel_handlers : 'a cancel_handler_set;
29    (* Functions to execute when this thread is canceled. Theses
30       functions must be executed before waiters. *)
31  }
```

The threads are pretty well commented within the code, however for brevity:

`Return`
: This state contains the returned value of thread. The return value is the result of the thread completing its execution successfully and terminating.

`Fail`
: The `Fail` state occurs when an uncaught exception is thrown from within the threads execution. The exception is wrapped around this `Fail` variant type so that another thread of execution may determine the appropriate next steps when dealing with this failed thread.

`Sleep`
: `Sleep` contains the `sleeper` type which holds all the main functionality of a thread. The `waiters` mutable state contains a list of all the functions that will be run when this thread is executed – it is the execution body of the thread. There is also state containing data which helps to cancel a thread, etc.

`Repr`
: The `Repr` state just holds the `thread_repr` type which is actually an indirection of a `thread_state`. Looking at the `thread_repr` type we can see that it simply holds another `thread_state` reference, which can be any of the above `Return, Fail, Sleep` or `Repr` variants. This allows chains of `thread_repr` objects pointing to a `Return, Fail` or `Sleep` thread at the end of the chain. This was an interesting problem to deal with whilst dealing with implementation discussed in 3.1.2.

Since LWT threads are cooperative, there's no real notion of spawning, starting

or stopping threads such as in POSIX or Java. Asynchronous execution is achieved by binding some work to a thread which will execute later on. This can be achieved with the `Lwt_main.yield` function:

```
1 val yield : unit -> unit Lwt.t
```

The `yield` function returns a thread which simply goes to sleep, then wakes up as soon as possible and terminates. This is fairly useless on it's own but binding to this thread will create a situation where you're able to schedule something as soon as a processor becomes available (which is the point at which the yield thread is woken up).

2.3 Investigating OCaml-Java

OCaml-Java, which provides the framework for compiling parallel OCaml code in this project, is a collection of Java programs and libraries that allow you to compile OCaml code to Java bytecode for execution on the JVM; run OCaml bytecode compiled software on the JVM using an interpreter written in Java; and access Java libraries directly from within OCaml code using a specially designed interface that maps Java classes to OCaml types. However most relevant property of OCaml-Java is it's ability to execute OCaml code in parallel using multiple Java threads.

OCaml-Java provides a concurrency module which contains many standard concurrency features such as locks, atomic variables and condition variables. The locks, for example, can be constructed by calling `make_reentrant` and manipulated by calling `lock` and `unlock` to change acquire/release the lock respectively:

```
1 val make_reentrant : bool -> t
2
3 val lock : t -> unit
4
5 val unlock : t -> unit
```

In fact, the concurrency module is just a set of OCaml wrappers around their Java counterparts[3] for example the OCaml-Java reentrant lock module `Concurrent.Lock` is a wrapper around the Java reentrant lock class `java.util.concurrent.locks.ReentrantLock`.

For classes that do not yet (and may never) have an OCaml wrapper, there is a neat interface which allows direct calls to Java functions and manipulation of Java references. Via the Java.make and Java.call functions, you are able to instantiate any Java class or call any Java function. An example which uses this OCaml-Java interface is given below:

```
1 let obj = Java.make "java.lang.Object()"
2 let itg = Java.make "java.lang.Integer(int)" 123l
3
4 let obj_hash = Java.call "java.lang.Object.hashCode():int" obj
5 let eq = Java.call "java.lang.Object.equals(java.lang.Object):
    boolean" obj itg
```

This code is an example of using the OCaml-Java interface to construct two Java objects named `obj` and `itg` using different classes via the `Java.make` function. Their hash values are compared to each other using `Java.call` on the `hashCode` method and the `equals` method. Whilst the Java primitives are easily mapped to OCaml types – e.g. a Java `int` maps directly to an OCaml `int32` – the types of Java objects need to have their own defining types within OCaml in order to cooperate appropriately with the OCaml type system. The typing scheme introduced maps a Java Object to an OCaml type `'a java_instance` where `'a` is a Java class. Classes are denoted within OCaml by replacing dots with single quotes to abide by the OCaml syntactic rules[3], therefore as an example, the equivalent OCaml type for `java.lang.Object` would be `java'lang'Object java_instance`.

2.4 Choice of Tools

My project depends entirely on the use of OCaml and Java therefore there is no real decision relating to the programming languages I will use. Furthermore the LWT library and OCaml-Java compiler are two projects that I will be depending on throughout the development of the project. These invariants aside, the following section outlines my choice of development environment, including version control, backup precautions taken/software used and the development and build methods applied.

2.4.1 Development Environment

Software and Systems

Unlike Java, OCaml has little in the way of a fully supported integrated development environment (IDE). There are add-ons for IDEs such as Eclipse[3] which add syntax highlighting and build functions, however most (including my supervisors) use a text editor such as Emacs or Vim to develop – which both come stock with OCaml syntax highlighting. I am already very familiar with Vim, therefore it was a clear choice to use it. I also used Mac OSX as my host operating system as I am a mac user, however, since there are little restrictions to what operating system OCaml (and in particular Java) can run on therefore I was also able to utilise the Intel Labs PWF machines on occasion which are running Ubuntu.

Version Control

For version control I used Git since I'm already familiar with it and furthermore it has always provided me with the necessary tools I need to work effectively and efficiently. I also used Github to host my code on a remote server for a few reasons:

- Github provides educational users with free private repositories which I used for this project.
- Storing my code on a remote server means that I can easily access it from anywhere with internet access, with all my revisions.
- My supervisors also have accounts on Github meaning they are easily able to check my progress.

Backup

In conjunction with Github as my primary backup solution, I also hosted my code on Dropbox in the case I lose work I haven't yet committed. Furthermore I also have mirrors of my Git repository hosted on the PWF facilities which were pushed to frequently.

2.5 Summary

Within this chapter I have described the process of planning and investigation taken in order to prepare for the projects implementation. I have investigated the

[3]https://www.eclipse.org

ways that OCaml and LWT are able to perform asynchronous code execution, including the cause for OCaml's limitations in fully parallel code compilation and the differences between LWT and Java's handling of threads. Furthermore I have demonstrated the tools that OCaml-Java has introduced which allow parallel OCaml code to be written and run on the JVM. Finally I explained that Vim will be my choice of IDE, running on Mac OSX; Git and Github will be used for version control and in addition the PWF facilities and Dropbox will be used for backup.

Chapter 3

Implementation

In this chapter, I outline the main directions and considerations taken towards developing the asynchronous framework for LWT on OCaml-Java. These are outlined as follows:

1. *Single Global Lock Approach*
   A single global lock is required to be held by any one worker thread (described below) that needs to execute a *lightweight thread*. This means that only one worker thread may be executing a lightweight thread at a time – the other threads are blocked and waiting on this lock. Whilst this method doesn't fully exploit parallelism, it provides a stepping stone in the process of constructing the worker threads and locking logic required for full parallel processing in addition to providing an earlier stage to see and record results.

2. *Coarse Grained Approach*
   This section extends the basic functionality of the runtime lock approach to multiple cores by introducing Java *worker threads*. These threads are assigned lightweight threads to work on continuously until the entire program has completed. Another very important consideration is to ensure that the worker threads are able to cooperate correctly with each other such that each lightweight thread may run in isolation, as if it were running in its native single core, asynchronous environment. In some ways that LWT is written assume (correctly) that it will be running on a single processor, resulting in the half-baked thread problem outlined below. Therefore I needed to take these assumptions into account whilst constructing the parallel system.

3. *Profiling*
   Profiling the performance gains from introducing parallel computation involved choosing appropriate benchmarks and implementing them in an LWT and OCaml-Java compatible manner. Ultimately merge sort and recursive Fibonacci were implemented to utilise multiple processors, and this section details the design decisions and their functionality within the system.

3.1 Parallelism Overview

The structure that LWT follows in order to create an asynchronous framework ultimately determined what the necessary steps were in implementing the LWT – OCaml-Java interface. LWT achieves asynchronous execution by storing yield threads (mentioned in section 2.2) in a queue which will be referred to as the *yield queue* from this point onwards. The function `Lwt_main.run` runs an algorithm which takes an initial lightweight thread and continuously polls this passed thread for completion (i.e. the thread state is `Return a` for some `a`). If the lightweight thread has not yet returned a value, the algorithm iterates through the yield queue, executing all the threads contained within it until it reaches the top of the queue. The algorithm then re-polls the passed thread for completion again, and cycles through the process of executing threads on the yield queue and polling for completion until a value is returned.

The functions to be executed by the worker threads will most likely add more lightweight threads to the yield queue. These threads are added to the bottom of the queue and are not executed until the next polling iteration. Figure 3.1 describes the functionality of the yield queue graphically. My approach to creating a parallel model of the yield queue was to mimic the functionality of the `Lwt_main.run` function across many Java worker threads using OCaml-Java. These worker threads perform the process of taking threads off the yield queue and executing them. Listing 3.1 summarises the behaviour of a worker thread. Initially the OCaml thread (that is, the first thread executing the main OCaml code) calls `Lwt_main.run` with an initial (passed) lightweight thread to execute. This function now spawns multiple worker threads (dependant on the number of physical threads the system has available) which are set to continually run lightweight threads on the yield queue until interrupted. The OCaml thread begins work on the passed thread which will inevitably spawn other lightweight threads to be pushed onto the yield queue. Once the initial passed lightweight

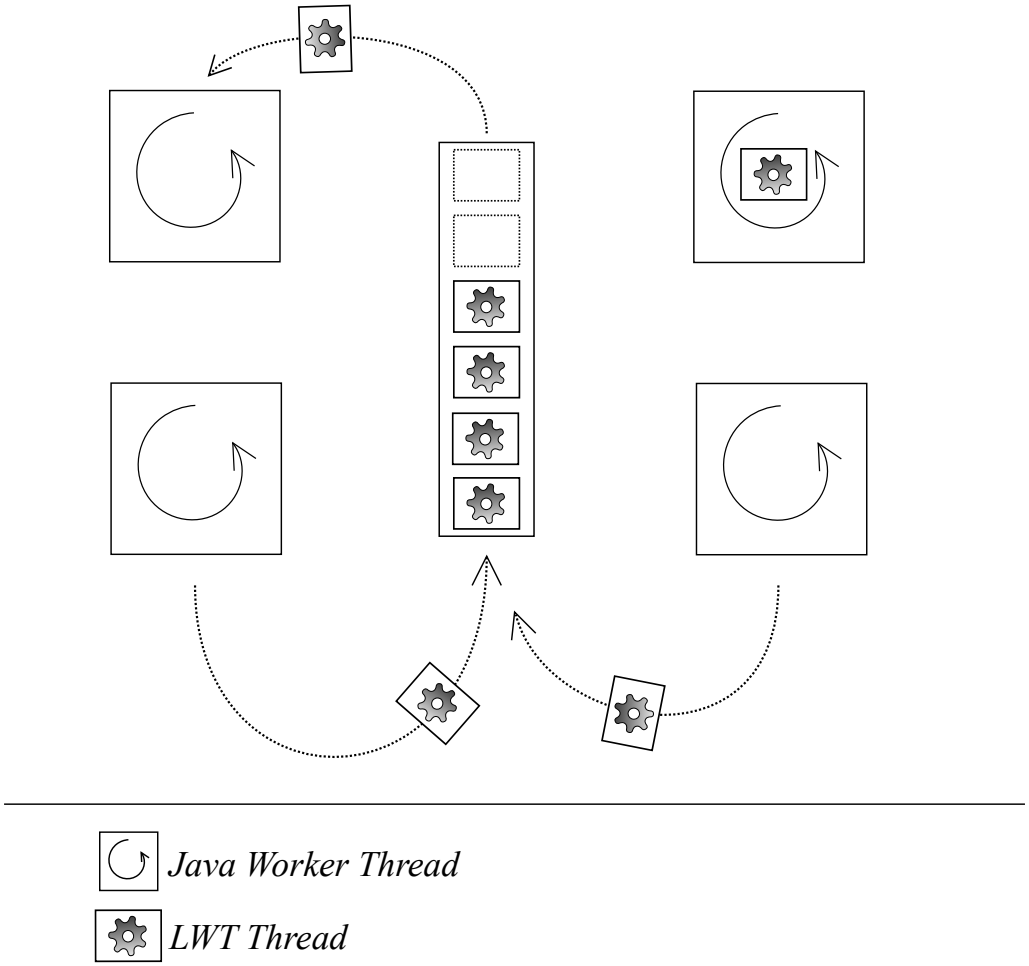

Figure 3.1: Visual demonstration of the process by which LWT threads are managed and executed on the worker threads. The diagram shows the two bottom worker threads placing LWT threads onto the yield queue, whereas the top left worker thread is removing an LWT thread from the yield queue and the top right thread is currently processing an LWT thread. Omitted from this diagram is the notion of the yield queue lock which ensures mutually exclusive access to the yield queue.

thread has finished work and is waiting on the spawned lightweight threads to complete, the OCaml thread will continuously poll the passed thread for a return value. Once this return value has been received, the OCaml thread sends an interrupt signal to all the other worker threads in order to stop processing yield threads and terminate execution. It is interesting to note that the OCaml thread is governing the other Java threads in a preemptive style whereas (as mentioned in section 2.2), LWT threads are cooperative – there were no problems working with the two threading ideas together.

Listing 3.1: Worker thread code

```
1  (* Function needs to lock the yielded list, pick up a sleeping
       thread and call 'wakeup' on it,
2       then rinse and repeat *)
3  let run_thread () =
4      while not (Thread.interrupted ()) do
5          Lwt_sequence.lock yielded;
6          let t = Lwt_sequence.take_first_l (fun t -> Lwt.
       try_lock_wakener t) yielded in
7              Lwt_sequence.unlock yielded;
8              match t with
9               | None -> () (* just spin round and try it again *)
10              | Some wakener ->
11                wakeup wakener ();
12                Lwt.unlock_wakener wakener
13     done
```

As an example, listing 3.2 is code that when run, simply outputs what thread it is running on. It does this 20 times before quitting, leaving a trace of all the threads that were executing at that time. The initial lightweight thread passed to `Lwt_main.run` is executed which spawns a lightweight thread that is placed onto the yield queue, furthermore another function (the waiter function) is bound to the lightweight thread such that when the thread is awoken and terminates, the function that was bound is then called with the return value of the lightweight thread. This function simply prints out what thread is currently running this particular code and then adds another lightweight thread to the yield queue, making itself the waiter function. This causes a continuous cycle where lightweight threads are added to the yield queue, taken off by some worker thread, and the name of the worker thread is printed out. There is a race between the worker threads as to which one picks up the yield thread, therefore the output from running this program may be different each time, demonstrating non-deterministic program behaviour.

Listing 3.2: Asynchronous Worker Thread example code

```
1  open Concurrent
2
3  let atomic_int = AtomicInt64.make (0L)
4  let seq_array = Array.make 20 ""
5
```

```
6  (* repeat determines whether atomic_int <= 19, then it sets
       seq_array [atomic_int] to it's thread name. This creates an
       ordering in seq_array of the threads access *)
7  let rec repeat () =
8  let x = AtomicInt64.increment_and_get atomic_int in
9    if (Int64.compare x 19L) <= 0 then begin
10     Array.set seq_array (Int64.to_int x) (Thread.get_name (Thread
       .current_thread ()));
11     Lwt.bind (Lwt_main.yield ()) repeat
12   end
13   else
14     Lwt.return ()
15
16 (* main function starts the first iteration of the repeat
       function *)
17 let () =
18   Lwt_main.run (Lwt.bind (Lwt_main.yield ()) repeat);
19   Array.iter (fun x -> print_endline x) seq_array
```

With the overview of the idea for implementing parallel thread execution in LWT, my approach took two main paths: a naive approach of using a single global lock and a coarse grained locking approach. I decided upon these two strategies because I considered the first approach to be a relatively straight forward way of getting a result from the project whilst also giving insight into potential problems. The second approach was partly based on the concurrency options that OCaml-Java provided: OCaml-Java provides reentrant locks and semaphores amongst the atomic variables within the concurrency module. Although I had initially decided to use semaphores with integer values based around the number of threads currently on the queue, as long as there is code running there will be threads on the yield queue because every call in LWT results in a thread. Implementing semaphores would cause more overheads with little returns, therefore a simple locking scheme was a more sensible idea in this respect.

3.1.1 Single Global Lock Approach

This approach involved having all worker threads require the 'global runtime lock' before attempting to pull anything off the yield queue. The lock is also held whilst the lightweight thread is being processed and is only released when all processing has been finished and the worker thread is about to enter its next iteration of work. All other threads are blocked, waiting on the runtime lock. Although each of the worker threads at this point are running on separate cores,

there is only ever one thread actually running at a time. Furthermore there is now an additional (although minimal) overhead of acquiring the single global lock before execution can proceed. However this approach has created a foundation for a more efficient method to be implemented.

3.1.2 Coarse Grained Locking Approach

Substituting a single global lock for a coarse grained locking implementation allows the project to take advantage of the parallelism exposed by OCaml-Java. The aim of the coarse grained lock is to prevent the unnecessary blocking of threads demonstrated in the single global lock approach, by keeping all the worker threads busy via an appropriate scheduler.

Lightweight threads in LWT have mutable state. Consider a lightweight thread state going from `Sleep` with no data to `Return` which lets anything accessing the thread know that the thread has finished executing and has returned its value. It is necessary for the old `Sleep` thread state to be removed (and collected by the garbage collector) and the `Return` state to be created so the value can be accessed. Therefore the locking mechanism requires that only one worker thread may be accessing a state at a time to ensure atomic access to the mutable state data. Furthermore, since each lightweight thread may have only one state, the best decision was to associate a lock with each thread which is to be acquired by all functions that access the thread.

Lock Sharing

As briefly mentioned in section 2.2, lightweight threads can have the type `thread_repr`. This type contains only a reference to another thread which in turn could be a `thread_repr` type, which results in a chain of `thread_repr` objects linking together until some other thread type terminates this chain. At each point in the chain, there may exist some reference to that point from another function or closure, figure 3.2 depicts an possible `thread_repr` chain in memory. Although each of these `threadi_repr` objects are distinct from each other, they all refer to the same base thread object at the bottom of the chain, whether it be a `Return` or `Sleep`, etc. Therefore it is important to ensure the way the thread locks are used deals with this layering appropriately. In deciding what the best approach for implementing thread locks in this situation, I came up with the two following methods:

1. Apply a different lock at each individual level, ensuring all locks are acquired and released at all levels below the current 'thread_repr' level before manipulating the thread data.

2. Have a lock at the lowest level only and ensure this is shared correctly between all other layers.

With method 1, it would be fairly easy to implement the lightweight thread locks as you would simply make a new lock for each `thread_repr` object, and apply a recursive lock down and throughout the chain. If another worker thread wishes to use the lightweight thread, they will begin to lock at some point within the chain but then reach a lock which they couldn't acquire. At this point the worker thread could either release its locks and perform some other useful processing or block on the locked thread.

Method 2 would is also relatively easy to implement but would require sifting through the various LWT ensuring that locks remain connected to their appropriate threads when dealing with mutable state, etc. However sharing the lock means only one lock operation would need to be performed at any level in the chain which is much more efficient than method 1. For this reason I chose method 2.

Thread Safety

Half-Baked Thread Problem Some aspects of how LWT handles threads did not directly map well to a parallel implementation of the interface. In some cases this meant that special care was taken to handle events in an appropriate manner. One notable example of this is a race condition between lightweight threads being placed on and taken off the yield queue before being appropriately initialised.

A visual description of the problem is outline in figure 3.3. The problem occurs when attempting to bind a lightweight thread to a yield thread so that it may be run asynchronously. The yield function adds a thread onto the yield queue instantly without the waiters being added to a waiter list. This means that any worker thread is able to remove the yield thread from the queue and run it. At the point the bind has added a waiter function to the waiter list, the yield thread has finished executing and will therefore never run the waiters now attached. Finally the garbage collector picks up the yield thread which has finished and it is never to be seen again.

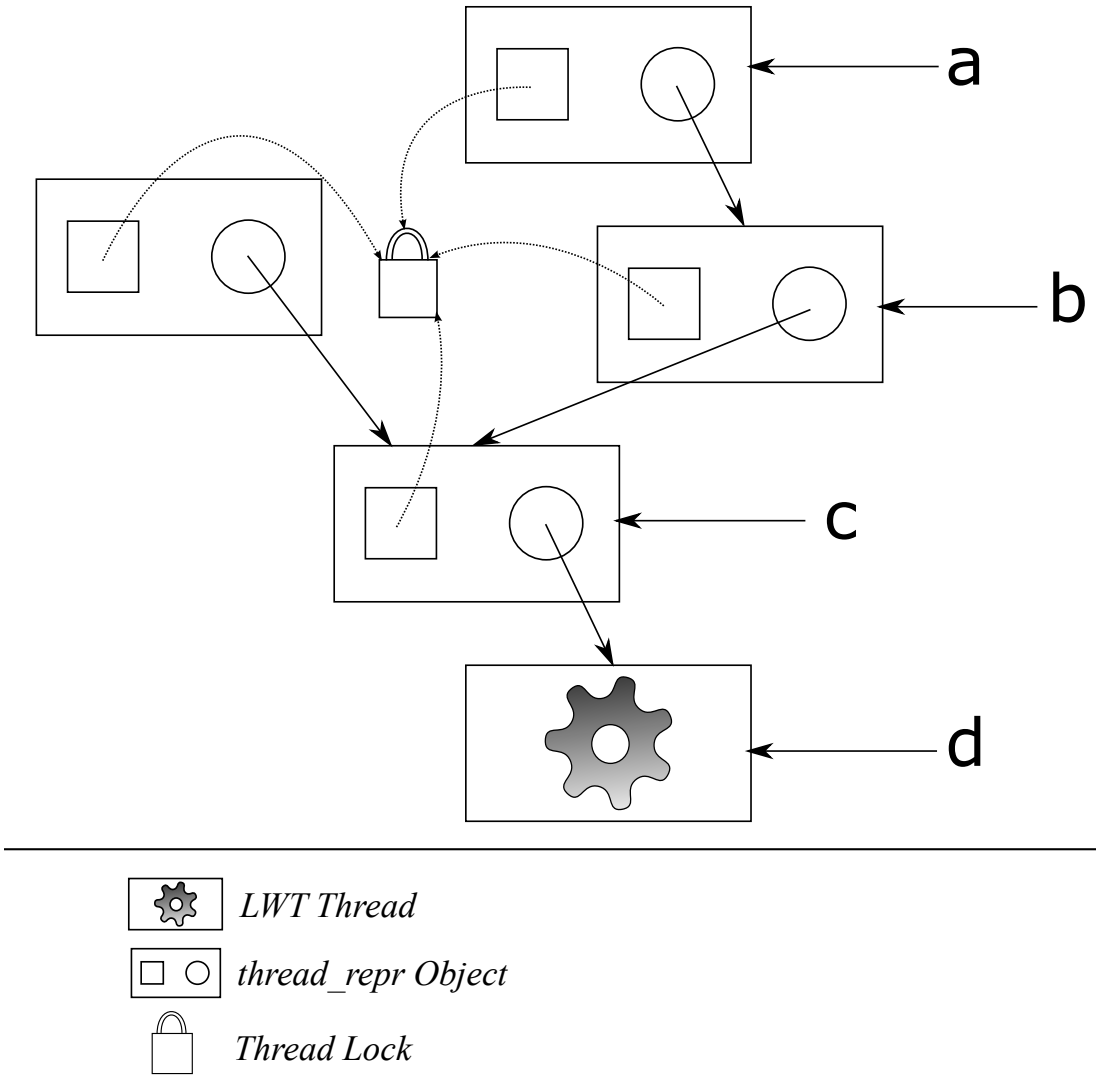

Figure 3.2: The diagram shows an example of an lightweight thread at the bottom of a chain of thread_repr objects. The letters to the right of each thread_repr object and the lightweight thread is a reference to the object from within the program. There may be access at any point within the chain during execution, therefore it is important for every thread_repr object to share the same lock – to prevent two worker threads using the same lightweight thread from different points in the chain. The links between thread_repr objects always create a directed acyclic graph which is why locking method 1 will always cross a lock which has been acquired if any locks have been.

Although the section of code where this race condition occurs is relatively small, the problem occurs surprisingly frequently. To overcome this issue, it was necessary to lock the yield thread between the point of creation and the end

of the bind. However this fix makes an assumption on how the yield thread is used – it suggests that a yield thread will only ever be used in conjunction with a call to bind. While this isn't strictly true, since a yield thread is a fully qualified lightweight thread on its own that simply does nothing, justification to this assumption is that – to my understanding – it is only ever sensibly used in conjunction with bind. To create a yield thread which sleeps and wakes with no functions to run upon termination is – useless. Furthermore the only way to achieve some kind of useful processing with a yield thread is to attach a waiter function which will run when the thread terminates.

To lock and unlock the yield thread separated across two functions, I used an additional thread value containing a possibly empty slot for a bind callback. A thread using bind which requires post-bind processing can fill this slot value with a function that performs the appropriate post-processing. Bind, upon completing its binding operation, will then execute the callback if it exists, finishing off the thread processing appropriately. Using this new bind callback, the call to yield first locks the yield thread before creation and adds a bind callback function which unlocks the yield thread. In this state, no worker threads will remove a lightweight thread on the yield queue which is currently locked (a requirement for removing a thread from the yield queue is that the worker thread is able to acquire the thread lock). Once the bind operation has completed on the yield thread, the callback is run which unlocks the thread, allowing it to be picked up by a worker thread to begin execution.

ACID Callbacks Callbacks in parallel processing can be called by any and all worker threads that have access to the code and shared data at any time, therefore it was also important to rewrite the LWT callbacks such that each thread accessing the callback does so in isolation from any other concurrent thread, the shared data is accessed atomically, the shared data remains in a consistent state and the shared data changes remain – which are the so-called ACID properties of database transactions. For two out of four of the ACID properties (consistency and durability) LWT's code already manages well, however atomicity and isolation issues needed to be dealt with. This mainly consisted of converting shared data to their atomic type equivalent, and ensuring data dependencies were kept serialised.

3.2 Profiling

Assembling an appropriate profiling suite to test the newly implemented functionality of OCaml-Java and LWT was a step as important as the interface implementation. The regular OCaml compiler allows a special flag for compiling native code with gprof profiling compatibility[13], however this is clearly unavailable since we are compiling to Java bytecode using OCaml-Java. There is no one single Java profiler tool such as gprof, however there are many open source alternatives. These are highly configurable and complex software suites which allow a lot of configuration – most of which I wouldn't need. For the purpose of profiling this project, I would simply require an accurate timer to determine the time taken to complete a function so that these times may be compared and the values used in speedup calculations in chapter 4. Taking this into consideration, I wrote a very simple timer module in OCaml using Java's high-resolution time source via `java.lang.System.nanoTime`. In fact, determining function running time is a typical use of the nanoTime method, as shown by the fact that within the Java documentation example code is given for exactly that purpose. Listing 3.3 shows some of the commands used within the simple timer module.

Listing 3.3: Timer module function types

```
1  (* timer type *)
2  type timer
3
4  (* make new timer *)
5  val make : unit -> timer
6
7  (* start timer *)
8  val start : timer -> unit
9
10 (* stop timer *)
11 val stop : timer -> unit
12
13 (* get time difference *)
14 val time_nanos : timer -> int64
15
16 (* get time difference *)
17 val time_micros : timer -> int64
18
19 (* get time difference *)
20 val time_millis : timer -> int64
21
```

```
22 (* get time difference *)
23 val time_secs : timer -> int64
24
25 (* reset timer *)
26 val reset : timer -> unit
```

To utilise Java functions and references directly through OCaml, I used the OCaml-Java interface described in section 2.3. There is no documentation on the speed of executing Java functions via the interface in comparison to direct calls, therefore it is possible that the interface will introduce some additional overheads, however they did not interfere with the results of the project as demonstrated in chapter 4.

Choosing an appropriate set of benchmarks which exploits parallelism well was important to demonstrate the true capabilities of my interface. To aid my choice I looked at the Barcelona OpenMP Task Suite and their choices of parallel benchmarks. One of their benchmarks is an integer sort algorithm based on the Cilk sorting algorithm. The Cilk project is a suite for algorithmic multithreaded programming and the Cilk sort performs a merge sort at the parallel level and an insertion sort at the sequential level, where sequential sorting is triggered by reaching a sub array size below a particular threshold. Furthermore the classic Introductions to Algorithms book [5] uses merge sort as an introduction to parallel algorithms. Therefore the most sensible choice for parallel benchmarking was the parallel merge sort algorithm due to its use in use in professional parallel benchmarking suites and major texts. The merge sort algorithm I implemented in OCaml is a mirror of the Cilk algorithm with the exception that the insertion sort in Cilk has been replaced with a simple call to OCaml's default array sorting algorithm `Array.sort`.

The merge sort algorithm within the system will continuously add merge sort commands for sub arrays onto the yield queue, where each sub array is half the size of its originating array. This will continue until a sub array drops below the linear sort threshold, at which point OCaml's default array sorting algorithm is called on the sub array. The threads are taken off the yield queue and executed on multiple worker threads as explained in section 3.1.

Another choice of benchmark which was both easy to implement and also featured within the Introduction to Algorithms book was the recursive Fibonacci algorithm adapted for multithreading.

3.3 Possible I/O Extensions

LWT has a very large collection of I/O modules which provide asynchronous I/O functionality and bindings to both Unix and Windows. This set of modules adds a generous amount of functionality to the core asynchronous library provided by LWT. Now I have converted the core library to work in parallel with OCaml-Java, it is also possible to adapt the LWT I/O modules to also utilise the Java I/O classes to provide asynchronous I/O to the system as well.

My approach to this problem would be in three stages:

1. Attempt to compile LWT in its entirety with OCaml-Java.
2. Filter out external functions that will not run.
3. Build up functionality in stages beginning from the filtered functions.

Attempting to compile LWT in it's entirety is a difficult job in itself as LWT relies on other packages such as OCamlbuild[1] and Findlib[2], which are not fully supported by OCaml-Java. The most straight forward option to compiling LWT would be to manually extract the build commands from OCamlbuild and output them to a custom build script, which performs the same standard job from a clean build directory. The Findlib package acts as an interface which allows easy package linking and more when building OCaml programs. Findlib outputs commands which contain all the relevant switches and linked files to compile a piece of OCaml software correctly, therefore by intercepting these commands and replacing them in the build script, I can remove Findlib and OCamlbuild support entirely. Furthermore LWT also uses the CamlP4 processor to provide many very useful shorthands when writing asynchronous LWT code. However there are a few bugs with the OCaml-Java Camlp4 system, therefore these files can be manually preprocessed using the default Camlp4 preprocessor which will produce the same output.

The second stage involves filtering out foreign function calls that will not run under the JVM. Examples of these commands are calls to system semaphores and mutexes, file manipulation commands and more, as they are all mostly system dependent. OCaml-Java does not support any foreign function calls written in C/C++, in fact it expects these external command calls to link to Java functions, existing in Java class files. Removing these external function

[1] OCamlbuild – http://caml.inria.fr/pub/docs/manual-ocaml-400/manual032.html
[2] Findlib – http://projects.camlcity.org/projects/findlib.html

calls from the OCaml code can be most easily done by replacing the calls with exceptions of the same command name – done with a sed script. This with both allow the code to compile and also make the third stage of the I/O interface creation process much simpler.

The third stage involves actually converting the external C/C++ commands into Java code so that it may be executed with (roughly) the same effect as it's C/C++ counterparts. At this stage of the conversion process, the LWT code should be compiling, however no I/O should yet be working but rather should be terminating with exceptions at the external calls. Depending on what code that I would like to have working first, I would begin with a test LWT program that utilises I/O, compile it and follow the exceptions, implementing each function until the software executes without fail. The idea is that this would create sections working I/O so that results may be seen sooner than later. An alternative approach would be to just attempt to implement all the functions, followed by testing and debugging – however the LWT codebase is large and being unable to test the functions (unable to test easily) would mean there is more code to sift through during the testing phase. The former approach is far more akin to an iterative/agile development process which is far more flexible to work with.

3.4 Summary

In this chapter I began with an overview of how parallelism is achieved between OCaml-Java and LWT, which uses worker threads that are fed lightweight threads off a yield queue, and explained that the rest of the chapter details the production of this system in it's different approaches. The first approach explained was the single global lock approach which acted mostly as a stepping stone into a stage where LWT was running in parallel on the JVM, whilst abstracting away the problem thread safety. This was a sensible method of dividing the task of implementing the system as the tasks of attempting to get LWT working on the JVM and exploiting parallelism have minimal overlap. The coarse grained locking approach section described the transition from parallel poorly scheduled LWT threads, to LWT threads being scheduled appropriately to many simultaneously executing worker threads. This stage introduced thread safety mechanisms based around lock acquisition and also the unforeseen technical details on getting LWT threads locking across multiple `thread_repr` objects, and solving the half-baked thread problem. I also ensured all callbacks adhered to the ACID concurrency

properties. Finally the profiling section details the design decisions taken whilst deciding on an appropriate choice of parallel benchmarks to implement for testing purposes, as well as methods of profiling OCaml running under the JVM without the help of OCaml's default compiler flags and settings.

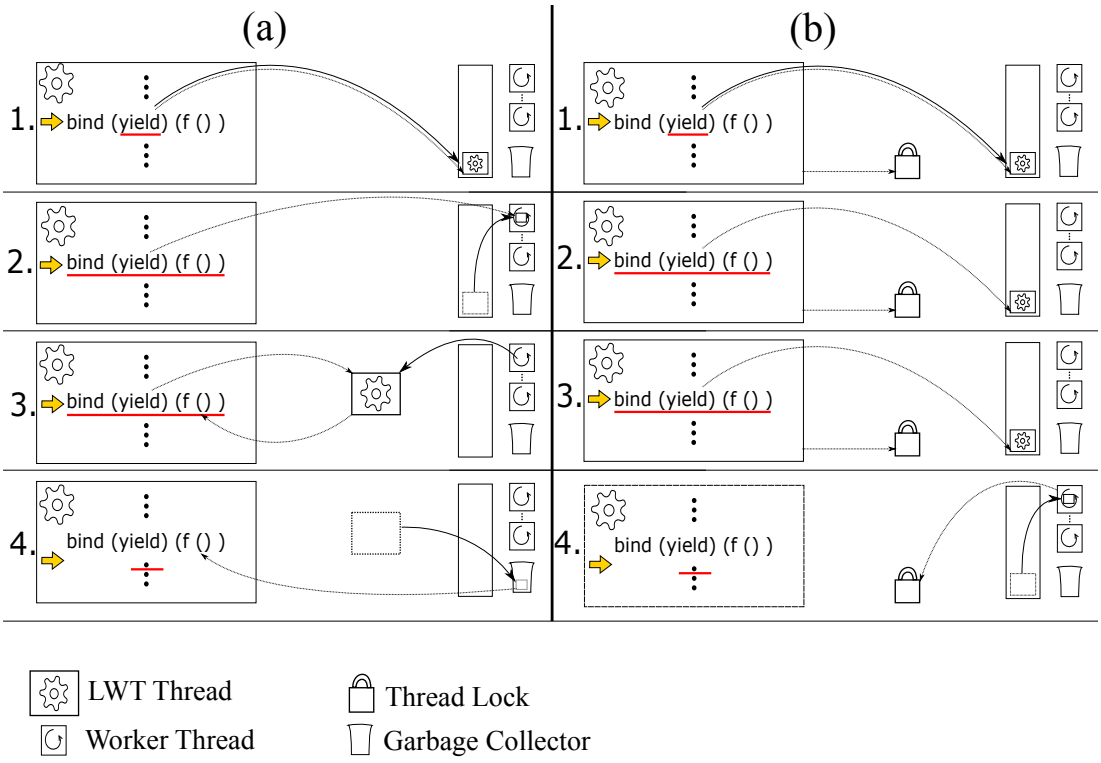

Figure 3.3: Figure (a) describes the process which causes the LWT thread to be exposed before it has finished processing. The dotted arrows represent references still held in memory and the solid lines represent movement of data. 1. The yield function is called and evaluated to a yield thread. This yield thread is placed straight onto the yield queue. 2. An idle worker thread picks up the yield thread and begins processing it whilst the bind function is processing more steps. 3. The worker thread has finished executing the yield thread and therefore it remains somewhere in memory. The yield thread is not collected by the garbage collector yet since there is still a reference to the data from within the bind call. Furthermore since the bind function has not completed at this point, the yield thread execution has not performed anything useful and has just in fact terminated. 4. The bind has now finished attaching a waiter to the yield thread that should execute when the yield thread is awoken, however this step has already occurred, therefore instead the yield thread is collected by the garbage collector and the function never executes. This is a classic Time of Check to Time of Use (TOCTTOU) *problem. Figure (b) shows the same 4 steps as figure (a) with the bind callback in place. 1. The yield function is evaluated to a yield thread and the locked yield thread is placed onto the yield queue. 2. Bind continues to execute but no worker threads are able to acquire the yield thread's lock therefore no worker threads will execute the yield thread. 3. Same as step 2. 4. Bind has finished and the next command begins executing. Bind has released the yield lock by calling the threads callback function which contains the* unlock code. *Now a thread is able to take the yield thread off the yield queue and begin execution with the correct waiters attached.*

Chapter 4

Evaluation

Within this chapter I discuss the merge sort test results which are graphed and the nature of the graph shape is explained with a proof that the speedup of parallel merge sort is $O(lg\, n)$, furthermore it is shown that this is consistent with the results obtained from a parallel merge sort written in Java. We also use the merge sort data do analyse the approximate threading overhead costs to discover how much processing power we are using on thread management. Fibonacci results are graphed and discussed, including how different implementations of the recursive parallel Fibonacci algorithm can achieve different results. I mention how the tests performed and the results obtained relate to my success criteria outlined in my project proposal. I also discuss how the tests were run, which consists of a small test suite deployed to the test machines I have been using.

The following evaluation demonstrates that my success criteria has been achieved. Below is a reminder of these criteria with a small summary of how each criteria has been achieved:

1. *I am able to show the scalability difference between two programs using the same piece of LWT code, one compiled using the original LWT libraries under the normal OCaml compiler and one compiled using my ported LWT libraries under OCaml-Java, running on the JVM.*
   The scalability differences between Fibonacci on normal LWT and my interface are demonstrated within this chapter in section 4.3.

2. *Parallel processing benchmark can be used to determine the scalability using LWT under OCaml-Java and using LWT under the normal OCaml environment.*
   The merge sort parallel processing benchmark scalability is also demonstrated in this chapter in section 4.2.

4.1 Testing

To perform the actual testing and result collection, I constructed a small test suite to automate tests both on my own machine and on Roo. Roo is a 48-core computer which I was provided access to by the labs in order to conduct my experiments. The relevant details of Roo's specifications, along with my own Macbook specifications, can be found in appendix A.1. This test suite consisted of functions which allow specific test parameters to be passed to the parallel benchmarks and software, such as the number of Java worker threads that must be created, the size of the merge sort array to be tested and other useful test parameters. The test suite also automatically determines whether some error occurred during testing by analysing the output and determining whether some tests should be repeated. The analysis does not test for anomalous results but rather incomplete/corrupted results. The help text is shown below:

```
Options Are:
    -f      Perform fib tests
    -m      Perform merge sort tests
    -F num  From value
    -T num  To value
    -S num  Step value
    -j num  Number of worker threads to be created in this test
            (not specifying will cause the test to repeat for 0
             <= j <= cores detected in machine)
    -n num  Size value
    -y num  Type of msort test to do, either msort varying array
            size (where num = 1) or sort varying linear sort
            limit (where num = 2). When type 1 is selected, the
            to, from and step values determine the range of
            array sizes to test (and the step size in-between),
            where the size value determines the size of the
            linear sort limit. When type 2 is chosen, the to,
            from and step values determine the linear sort limit
            value and the size value determines the array size to
            be used throughout the tests.
```

Without any parameters passed, the test suite simply runs the default tests which provide appropriate numbers for analysis.

4.2 Merge Sort

The parallel merge sort algorithm briefly mentioned in section 3.2 provides a good parallelism benchmark to test the interface with. Firstly, each recursive

call of the merge sort works on a completely independent sub section of the array from its sibling's recursive call. This means that (assuming a contiguous in-memory array is used) there is little/no data dependency between concurrent threads running merge sort and therefore they may run without stalling each other or requiring special locks. Furthermore, the Cilk benchmark which this current merge sort implementation is based on performs a linear sort when the array size drops below a certain threshold. By adjusting this value appropriately we are able to achieve the best performance gains by allowing LWT threads to perform large chunks of processing and less thread management. The pseudo code for the merge sort algorithm is provided below:

Apart from the actual sorting algorithm, there is the situation of creating appropriate arrays for the merge sort algorithm to test and sort. In fact the merge sort is provided a random list of numbers from a range equal to the length of the array: that is, an array of length 10 will contain only the numbers $1 - 10$ at random. This made specifying the range of numbers to the OCaml `Random.int` function easier. Furthermore the merge sort algorithm was to be run at least twice, once using LWT with no parallelism and one with parallelism, therefore I made copies of the arrays and provided the copies to each function so that they would work on the same, equal data. However the array's do change between different instantiations of the test, such as when changing the number of worker threads being tested.

4.2.1 Performance Analysis

Merge Sort Speedup

The performance results of the merge sort algorithm are shown in the speedup graph in figure 4.1, where the x axis gives the number of cores in use and the y axis is the speedup, calculated as T_1/T_n where T_1 is the time taken to perform the sort on 1 thread and T_n is the time taken to execute the sort on n threads. There is a clear, almost linear speedup between using 1 and 8 cores for the sort, reaching up to a maximum speedup of between approximately 3 or 4. However after this point, the graph seems to plateau with a little jitter in between successive increments in the number of cores.

We can first analyse the serial and parallel merge sort algorithms mathematically and then apply this to the outcome of the graph. It is a well known fact that the average running time of merge sort is $\Theta(lg\, n)$, this can be shown

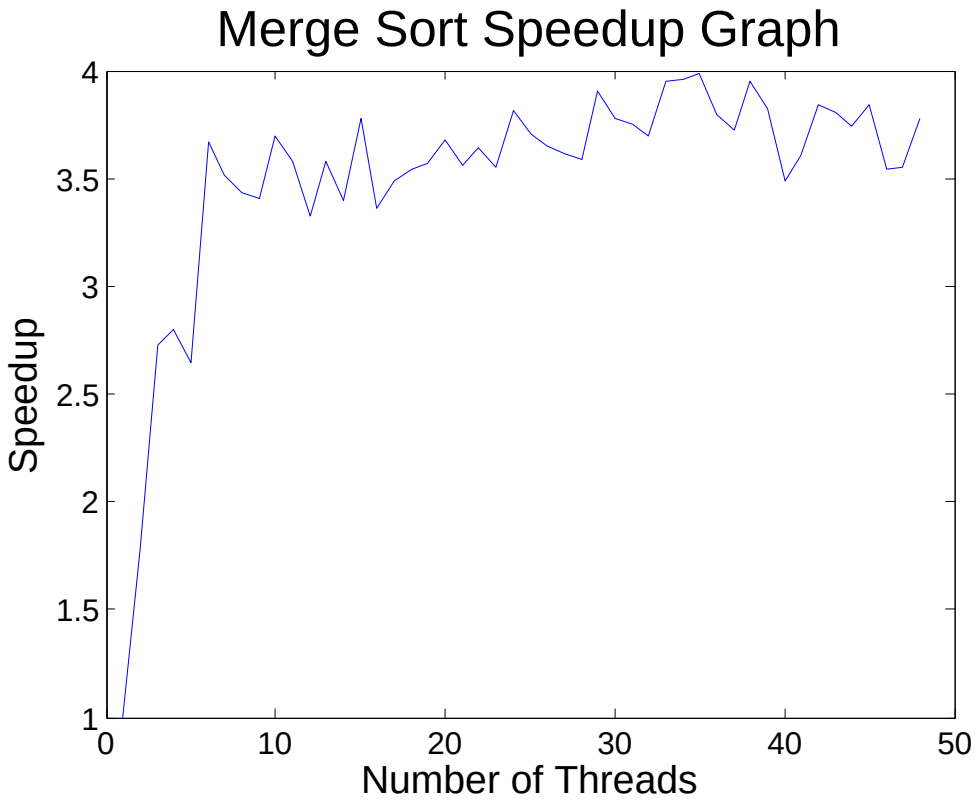

Figure 4.1: The merge sort speedup graph, plotted with a range between 1 and 48 cores on the x axis; speedup calculated as $\frac{T_1}{T_\infty}$ along the y axis. The graph shows a logarithmic curve which seems to plateau around 3.5 – 4 times speedup. The logarithmic shape is exactly what should be expected as the merge sort algorithm has maximum speedup in the order $\Theta(lg\, n)$ which is shown within section 4.2.1. This result is also mirrored in a normal Java merge sort speedup graph shown in figure 4.2. Interesting aspects of the graph are the jitter, which can be explained by the fact that the asynchronous nature of the algorithm means that the execution paths are slightly different each time, in conjunction with the fact that the arrays are generated randomly each time with only similar length therefore the array ordering may cause a faster or slower execution time.

by considering the following recurrence relation:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

This recurrence relation is constructed by dividing the merge sort into its component operations: *Divide, Conquer and Combine*[1]:

[1]Divide, Conquer and Combine is the way CLRS describes the separate stages of merge sort.

Divide

The dividing process simply splits the array into two smaller sub arrays. This is a cheap operation since all that is needed is to calculate the existing address of the sub array in memory.

Conquer

Conquering consists of performing a recursive call on the two sub arrays created in the divide process. If the time taken to perform a merge sort on an array of size n was $T(n)$ then the conquer process must perform this operation twice on two sub arrays of half the size, resulting in the term $2T(\frac{n}{2})$.

Combine

The combining phase is the *merge* of the merge sort algorithm, which runs linearly through the two arrays, creating a larger merged and sorted array, in time $\Theta(n)$ for sub arrays of size n each.

Solving this recurrence relation is relatively simple algebra with a little substitution:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \Theta(n) \\ &= 2\left(2T\left(\frac{n}{4}\right) + \Theta\left(\frac{n}{2}\right)\right) + \Theta(n) \\ &= 4T\left(\frac{n}{4}\right) + 2\Theta(n) \\ &= 8T\left(\frac{n}{8}\right) + 3\Theta(n) \end{aligned}$$

At this point if we let $n = 2^m$ we get:

$$\begin{aligned} &= 2^3 T\left(2^{m-3}\right) + 3\Theta\left(2^m\right) \\ &= 2^k T\left(2^{m-k}\right) + k\Theta\left(2^m\right) \qquad\qquad \forall k \geq 0 \end{aligned}$$

Letting k tend to m (approaching the smallest sub array size):

I believe this is very clear and sensible and therefore have done the same here

$$
\begin{aligned}
&= 2^m T\left(2^0\right) + m\Theta\left(2^m\right) \\
&= 2^m + m\Theta\left(2^m\right) \\
&= n + lg(n)\Theta(n) && \text{substituting } 2^m \text{ for } n \\
&= \Theta(n\, lg\, n)
\end{aligned}
$$

This is the average running time of a serial merge sort and also the work of a parallel merge sort (where work is denoted T_1), therefore this value is the running time it should take a merge sort to complete on a single core. Given the opportunity to utilise many cores, there are subtle changes to the occurrence relation. For my current implementation of merge sort, the merge operation still remains linear, therefore this term remains as $\Theta(n)$, however the recursive call is able to run on different cores simultaneously, therefore this term now becomes $T'(\frac{n}{2})$ where T' is the parallel merge sort time. Thus the recurrence relation looks like:

$$
\begin{aligned}
T'(n) &= T'\left(\frac{n}{2}\right) + \Theta(n) \\
&= T'\left(\frac{n}{4}\right) + \Theta\left(\frac{n}{2}\right) + \Theta(n) \\
&= T'\left(\frac{n}{8}\right) + \Theta\left(\frac{n}{4}\right) + \Theta\left(\frac{n}{2}\right) + \Theta(n)
\end{aligned}
$$

Similar to last time, let $n = 2^m$:

$$
\begin{aligned}
&= T'\left(2^{m-3}\right) + \Theta\left(2^{m-2}\right) + \Theta\left(2^{m-1}\right) + \Theta(2^m) \\
&= T'\left(2^{m-k}\right) + \sum_{r=0}^{k-1} \Theta\left(2^{m-r}\right) && \forall k \geq 0
\end{aligned}
$$

Let k tend to m:

$$
\begin{aligned}
&= T'\left(2^0\right) + \Theta\left(2^1\right) + \ldots + \Theta\left(2^m\right) \\
&= 1 + \Theta\left(2^1 + 2^2 + \ldots + 2^m\right) \\
&= \Theta\left(2^{m+1} - 1\right) \\
&= \Theta\left(2 \cdot 2^m\right) = \Theta\left(2n\right) = \Theta\left(n\right)
\end{aligned}
$$

Taking into consideration the ability to run the recursive calls on many processors, we have calculated the parallel merge sort average running time, which is also the span of a parallel merge sort (span denoted by T_∞). So currently we have the results $T = T_1 = \Theta(n\,lg\,n)$ and $T' = T_\infty = \Theta(n)$, for which we can use to calculate the theoretical speedup between serial and parallel merge sort:

$$\text{Speedup} = \frac{T_1}{T_\infty} = \frac{\Theta\,(n\,lg\,n)}{\Theta(n)} = \Theta(lg\,n)$$

Therefore in fact we are expecting to see a logarithmic shaped graph as our speedup graph, which is exactly what figure 4.1 shows us. Figure 4.2 shows a parallel merge sort written in Java[17] with the same test parameters and the speedup results obtained from these tests which mirrors the results obtained from using the interface. This is a very good sign to suggest that the interface is performing well. The normal Java merge sort reaches a peak speedup of around 5 whereas the interface provides a peak speedup between 3 and 4 which is a confident result for both the OCaml-Java compiler and the interface.

Thread Scheduling Overhead

The last section dealt with understanding the speedup properties of the merge sort algorithm and comparing that to the test results achieved. One important aspect that wasn't taken into account is the overhead caused by the scheduler and how much that actually effects the computation time of the merge sort algorithm. We can determine the threading overheads in terms of the execution time of the merge sort by alternating the linear sort limit of the merge sort algorithm – which is the maximum array size that will be sorted on one thread, i.e. linearly.

For a smaller linear sort limit, the merge sort algorithm does less computation per core and more lightweight threads are created as there are more divisions of the array to be made before the linear sort limit is reached. However at some point we expect the lightweight threads to be sorting significant portions of the array such that the threading overheads are negligible. Increasing the linear sort limit again beyond this point will result in a longer execution time as the algorithm will not divide the data up enough to keep all worker threads continuously busy. It is the point at which the lightweight thread overheads become equal to the work done by the linear sorts which is important to us. At this point we can divide the execution cost roughly between thread scheduling and linear sort execution times in order to calculate the average overhead per thread. The point at which this occurs is roughly the of the point at which the

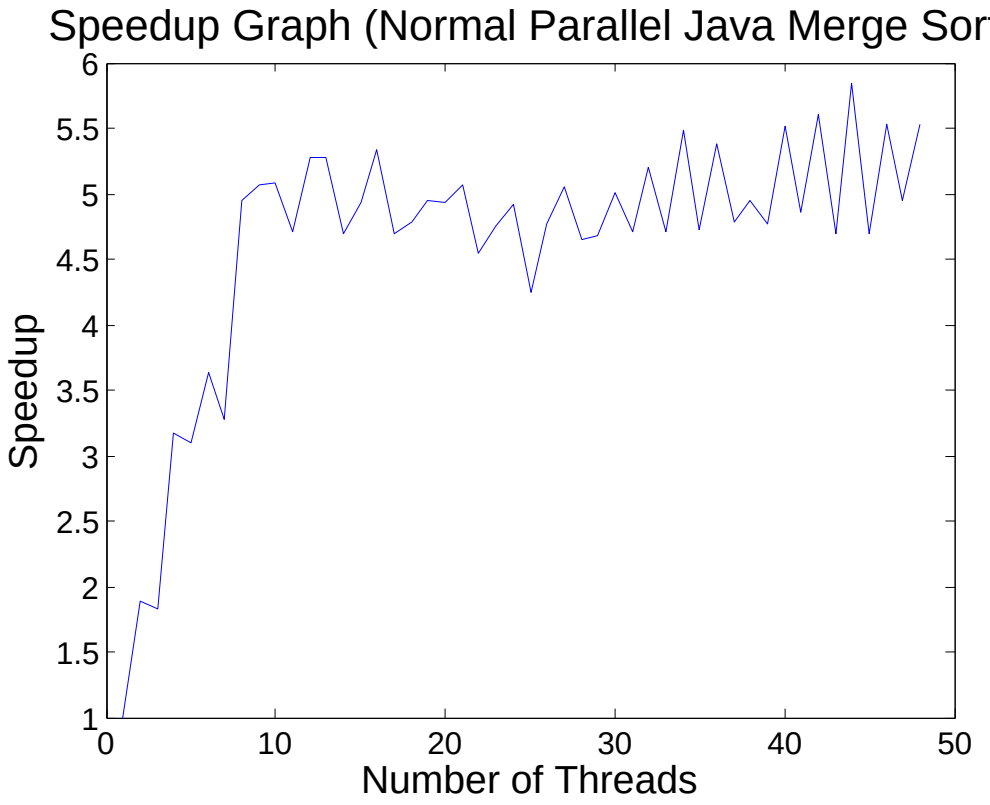

Figure 4.2: The Java speedup graph, showing a merge sort written in Java, mirrors exactly what we have achieved by using the interface. The speedup in this graph plateau's at approximately 5 – 5.5 which slightly greater than the interface's merge sort results as would be expected since the interface constructed on a few layers and therefore will omit more overheads. This graph also includes a little jitter which is again due to the changes between the array's values in between each test.

speedup gains from increasing the linear sort length begin to plateau. Figure 4.3 shows the test results collected from increasing the sort limit along the x axis for an array size of 30 million.

It's clear to see the beginning of the plateau occurs around 8 on the x axis, which being a log_2 scale corresponds to a sort limit of 256 values. The total amount of threads created at this point can be calculated as:

$$threads = \lceil 2^{log_2(30{,}000{,}000) - log_2(256)} \rceil - 1$$

Since the height of the merge sort array division tree is $\lceil log_2(30,000,000) - log_2(256) \rceil = h$ and the number of threads equals the number of leaves within the tree, calculated as $2^h - 1$. Finally we simply need to divide half the execution

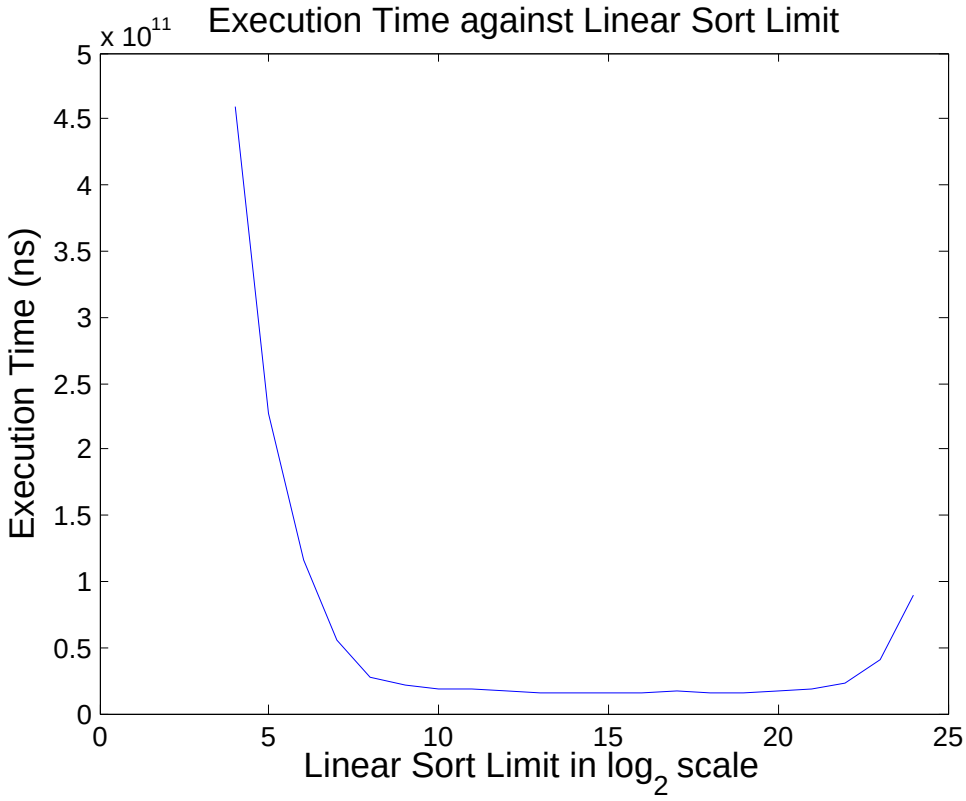

Figure 4.3: Sort limit graph showing the logarithmic increase in the linear sort limit on the x axis along with it's effect on execution time on the y axis. There is a steady decrease in execution time as the sort limit increases between 16 and 256, (4 and 8 on the x axis). After this point the execution time seems to plateau until 2^{22} linear sort limit at which point the execution time begins to rise again. The initial decline is caused by the worker threads performing more processing per thread which decreases the execution time. At the point of the plateau, the amount of work done by each thread is significant enough to keep all cores busy and the threading overheads negligible, therefore we do not see any change in the execution time. The final rise at the right of the graph occurs as the sort limit tends towards the size of the array, so in fact the array is not being divided up enough to keep all the cores busy.

time by the total amount of threads to get the average overhead per thread in nano seconds:

$$avg\,overhead = \frac{13426901208}{\lceil 2^{log_2(30{,}000{,}000) - log_2(256)} \rceil - 1}$$
$$= 0.11\,ms$$

That's quite a long overhead for a lightweight thread, however considering

that OCaml-Java itself is approximately 3 times slower in executing natively compiled OCaml code[3] the large overhead was to be expected. There are subtle ways to improve the overheads without interfering with OCaml-Java, however the speedup results of the parallel merge sort algorithm still show very positive results regardless of this thread overhead.

4.3 Fibonacci

Also mentioned in section 3.2 was the recursive Fibonacci algorithm for use in profiling the interface. The recursive algorithm is easily made parallel by spawning child threads at each recursive call until the base case is reached. Each thread spawned is placed onto the yield queue, until a worker thread is able to pick it up and either perform the base case calculation or process the next recursive step – creating more threads on the yield queue. The parent thread will then synchronise on both child threads completing and use those results to calculate it's own Fibonacci result. This propagates back up the Fibonacci recursive tree until the final result is calculated and returned to the initial caller function. The Fibonacci code used can be found in appendix B

4.3.1 Performance Analysis

Execution Time

The results of performance testing the recursive parallel Fibonacci algorithm are displayed in figure 4.4. This graph describes the execution time of the Fibonacci algorithm where the x axis denotes the Fibonacci number being calculated and the y axis denotes the time taken to calculate that Fibonacci number. The graph clearly shows that calculating Fibonacci on multiple cores actually performs worse than using the same recursive algorithm on a single core. This is highly counter intuitive under the assumption that more cores means more power. Although there are many more cores executing the Fibonacci calculations, it would seem the overhead of managing many LWT threads is significant enough to completely negate any potential speedup gains from utilising multiple cores.

At each stage in the Fibonacci calculation, a new thread is created for that recursive step. Once the thread is created, it is popped onto the yield queue and may sit there for a while until it is acquired by a worker thread to perform execution. Within the calculation, the fib thread must bind on the result of both it's child threads and perform an addition. This bind operation

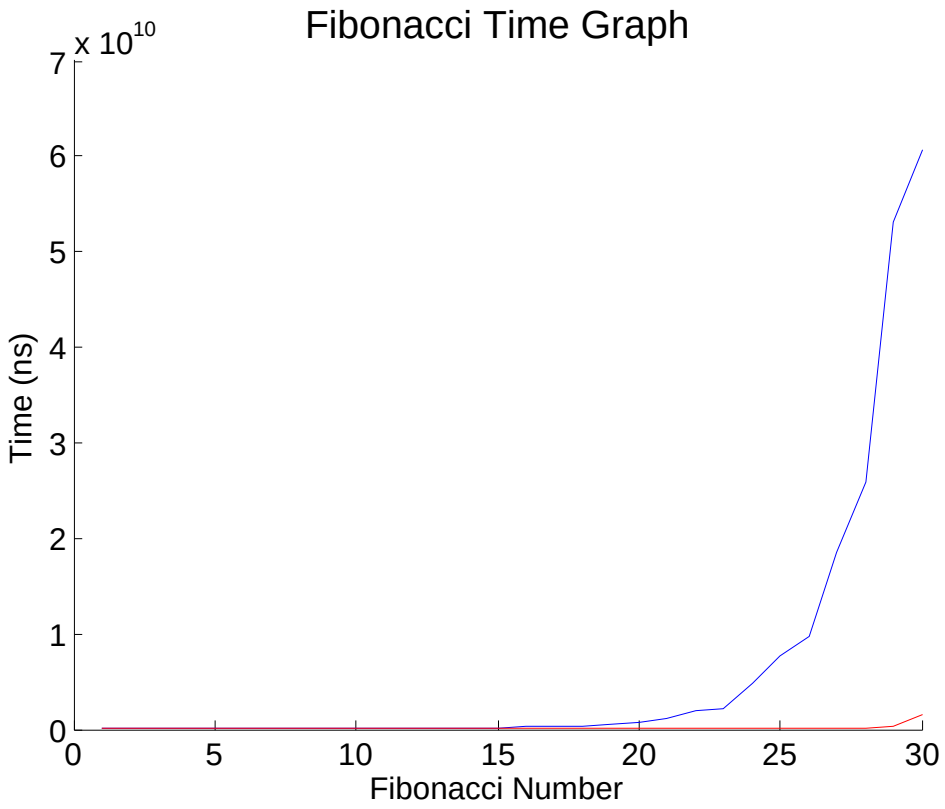

Figure 4.4: The steeply rising line represents the parallel Fibonacci algorithm whereas the shallow line represents the single threaded Fibonacci algorithm. It is clear from this graph that despite having many cores available, the threading overheads entirely overshadow the potential for speedup.

consists of creating a closure containing an addition operation and the two child threads, such that when they both complete, the closure will be called with their return values and pass the result back up the chain. We discovered in the last section that the average running time for all these steps is roughly 0.11 ms which is a relatively significant amount of time. Furthermore all these extra steps are produced to simply perform a single addition operation which means the majority of the processing done on the worker thread is the thread management rather than the algorithm processing.

Overall the Fibonacci calculation has not scaled well with the current implementation of the interface, unlike the merge sort. However it may be possible to achieve better performance gains by using a different scheduling technique than the one currently in use. Furthermore the current implementation of the

parallel Fibonacci continues to recurse until it reaches $fib(0)$ or $fib(1)$, by limiting the recursion depth of the Fibonacci algorithm in a similar fashion to the linear sort limit of the merge sort algorithm, we can combine more processing potential into a single lightweight thread and definitely achieve better results.

4.4 Summary

Within this chapter, I discussed how the merge sort algorithm was tested and also the performance of the merge sort algorithm when executed with the interface. The results obtained demonstrated the interface's true ability to utilise multiple cores and execute parallel LWT code with appropriate speedup gains, achieving a speedup of 3 – 4 in comparison to a normal Java merge sort algorithm which achieved a speedup of 5 – 5.5 with the same test parameters. The reason to the graph shape was proven as the theoretical speedup of the merge sort algorithm and was observed in both the interface parallel merge sort and the normal Java merge sort results. We also used the variation of the linear sort limit to determine the approximate thread management overhead of the lightweight threads, which resulted in 0.11 ms – a good result considering OCaml-Java's approximately 33% reduction in execution speed.

The parallel Fibonacci algorithm provided somewhat different results as we discovered the recursive Fibonacci algorithm mainly exposed thread overhead times since the algorithm continues to recurse until it reaches $fib(0)$ or $fib(1)$. However we determined that setting a recursion limit would move the majority of the processing time from thread management to algorithm execution in a similar way to the linear sort limit in the merge sort algorithm.

Chapter 5

Conclusions

5.1 Lessons Learnt

The final result of my dissertation has been successful. I have managed to complete all my success criteria and produce an interface which scales similarly to natively written Java code when comparing popular parallel benchmarks such as merge sort. Admittedly taking on such a large OCaml code base with only little prior knowledge of ML was a feat I under anticipated. The languages have a few differences, however the styles of development in the two languages are very independent of each other. A lot of the focus in learning ML was put on developing algorithms and data structures, whereas OCaml's Core library contains most of these algorithms and has an emphasis on using them instead. Furthermore in languages like C++ and Java there are structured and tested ways of producing systems with low coupling and high cohesion via object oriented programming techniques and design patterns, however in OCaml you can choose to use the modular system, the object system, neither or a mixture of both[2].

That being said, I've learnt a lot of important lessons about developing functional code and also project management. One of the main problems I had with this project was biting off more than I could chew. At times within LWT there was very high coupling between modules and I found myself attempting to juggle developing and converting code from many different locations in code at the same time in an attempt to keep the entire system working. Inevitably it served me better to attempt to reduce coupling as much as possible and focus on core sections of the code base, rather than as a whole. At times it was also characteristic of me to become 'lost in the code', where I would be thinking of

many other things I could do to better my project or produce cleaner code/tests. Whilst this is perhaps a good mindset to have when dealing with a project it can at times lead me astray from my targets and success criteria, therefore it was good to take a minute, step back and look at the picture as a whole: where I am right now, where I need to be and what I need to do to get there.

5.2 Future Work

In conjunction to the work already demonstrated within this project, I also have some ideas on how else some of the challenges in this project could have been overcome – it would be interesting to see how the performance changes depending on how the interface is implemented.

5.2.1 Public/Private Queues

In section 3.1.2 I mentioned the 'Half-Baked Thread' problem which meant that useless yield threads were being placed onto the yield queue and immediately removed by other threads before having any useful bindings attached. One other solution to this problem I have thought about was to have a separate private and public queue per worker thread. As a worker thread is executing some code within an LWT thread, it will place any created threads into its private queue, which may contain threads that are or are not able to be executed. No other worker thread may have access to the private queue of any other worker thread. Once an LWT thread has terminated, the private queue will contain the new spawned threads which resulted from the LWT threads execution, furthermore they will be in a state where they are all ready to execute. The worker thread must then move these ready threads to the public queue, all bar one, and begin execution of the one retained thread. By keeping all the threads in the private queue until the LWT thread has finished execution, there is no chance that another thread will come along and take the incomplete thread and begin executing it. Only threads on the public queue are able to be taken by other threads for execution.

Bibliography

[1] Jane Street Capital. Technology - Jane Street. http://janestreet.com/technology/.

[2] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Developing Applications with Objective Caml*. O'Reilly, 2000.

[3] Xavier Clerc. Ocaml-java 2.0. http://ocamljava.x9c.fr/preview/.

[4] Xavier Clerc. OCaml-Java: OCaml on the JVM. *Trends in Functional Programming*, pages 167–181, 2012.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2013.

[6] Microsoft Corperation. auto Keyword (Type Deduction). http://msdn.microsoft.com/en-us/library/vstudio/dd293667.aspx.

[7] Oracle Corperation. Garbage-first collector. http://docs.oracle.com/javase/8/docs/technotes/guides/vm/G1.html.

[8] Oracle Corperation. Lambda Expressions. http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

[9] Oracle Corperation. Learn About Java Technology. http://www.java.com/en/about/.

[10] Oracle Corperation. Thread Pools. http://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html.

[11] Jérémie Dimino. Lwt User Manual. 2012.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantative Approach*. Morgan Kaufmann, fifth edition, 2012.

[13] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, , and Jérôme Vouillon. The OCaml System Release 4.01. 2013.

[14] Anil Madhavapeddy, Jason Hickey, and Yaron Minsky. *Real World OCaml*. O'Reilly, 2013.

[15] Mathworks. Matlab - the language of technical computing. http://www.mathworks.co.uk/products/matlab/.

[16] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, third edition, 2009.

[17] Marty Stepp. Introduction to parallel algorithms. http://courses.cs.washington.edu/courses/cse373/13wi/lectures/03-13/.

Appendix A

Notes

A.1 Machine Specifications

The following are specifications for the machines used during the testing and development of the interface:

'Roo' Multi-core Machine - Main Test Machine

- *Processor*
  AMD Opteron™ 6344 Processor (12 logical cores, 6 physical cores @ 2.6 GHz) (×4)
- *Main Memory*
  64GB RAM

A.1.1 Macbook Air 2013 - Main Development Machine

- *Processor*
  Intel Core i7 (4 logical cores, 2 physical cores @ 2.6 GHz)
- *Main Memory*
  8GB RAM

Appendix B

Code Samples

The following appendix contains relevant code samples which have been referred to within my dissertation.

B.1 Recursive Parallel Fibonacci Implementation (fib.ml)

The recursive Fibonacci algorithm was implemented 3 times in different ways in order to compare the normal, asynchronous and parallel versions for execution time, speedup and other relevant measurements. The following code listing shows each of these versions.

Listing B.1: Fibonacci code

```
1
2  let rec fib_seq stack n =
3    match n with
4    | 0 -> 1
5    | 1 -> 1
6    | n -> let fib_left = (fib_seq ("left " :: stack) (n-1)) and
       fib_right = (fib_seq ("right " :: stack) (n-2)) in
7          fib_left + fib_right
8
9
10
11 let rec fib_lwt stack n =
12   match n with
13   | 0 -> Lwt.return 1
14   | 1 -> Lwt.return 1
15   | n -> let fib_left = (fib_lwt ("left " :: stack) (n-1)) and
       fib_right = (fib_lwt ("right " :: stack) (n-2)) in
```

```
16      Lwt.bind (fib_left) (fun fib_left ->
17        Lwt.bind (fib_right) (fun fib_right ->
18          Lwt.return (fib_left + fib_right)))
19
20
21  let rec fib_async stack n =
22    match n with
23      | 0 -> Lwt.return 1
24      | 1 -> Lwt.return 1
25      | n -> Lwt.bind (Lwt_main.yield ()) (fun () ->
26        let fib_left = (fib_async ("left " :: stack) (n-1)) and
       fib_right = (fib_async ("right " :: stack) (n-2)) in
27          Lwt.bind (fib_left) (fun fib_left ->
28            Lwt.bind (fib_right) (fun fib_right ->
29              Lwt.return (fib_left + fib_right))))
```

B.2 Parallel Merge Sort Implementation (msort.ml)

Similar to the Fibonacci implementation, the merge sort was also written in three different ways.

Listing B.2: Merge Sort code

```
1  (* Mutable integer record type *)
2  type mut_int = {mutable data : int}
3
4  (* Immutable helper functions *)
5  let rec take l num = match l with
6    | x::xs -> if num > 0 then x::(take xs (num-1)) else []
7    | [] -> []
8
9  let rec drop l num = match l with
10   | x::xs as xlist -> if num > 0 then (drop xs (num-1)) else
      xlist
11   | [] -> []
12
13 let rec len l = match l with
14   | [] -> 0
15   | x::xs -> 1 + (len xs)
16
17 (* largest array size to perform linear sort *)
18 let sort_limit = ref (1000)
19
20 (* function generates a random array of integers *)
```

```
21 let rand_array n max =
22   Random.self_init ();
23   Array.init n (fun _ -> Random.int max)
24
25 (* Merge sort with immutable datastructures *)
26 let rec merge_sort_imm l =
27   let rec merge a b = match (a,b) with
28     | ( x::xs as xlist , (y::ys as ylist) ) -> if x < y then x::(
    merge xs ylist) else y::(merge xlist ys)
29     | ( [] , ylist ) -> ylist
30     | ( xlist, [] ) -> xlist
31   in
32     match l with
33       | [] -> []
34       | x::[] as xlist -> xlist
35       | _ as xs ->
36           let length = (len xs) in let half_length = length / 2
    in
37             merge (merge_sort_imm (take xs half_length)) (
    merge_sort_imm (drop xs (length - half_length)))
38
39 (* imperative merge function (to be reused) *)
40 let merge a p q r =
41   let n1 = q - p + 1 in
42   let n2 = r - q in
43   let lhs = Array.sub a p n1 in
44   let rhs = Array.sub a (q+1) n2 in
45   let i = { data = 0 } in
46   let j = { data = 0 } in
47   let k = { data = p } in
48     while (i.data < n1) && (j.data < n2) do
49       let lhs_elem = Array.get lhs i.data in
50       let rhs_elem = Array.get rhs j.data in
51         if lhs_elem <= rhs_elem then begin
52           Array.set a k.data lhs_elem;
53           i.data <- i.data + 1;
54           k.data <- k.data + 1;
55         end
56         else begin
57           Array.set a k.data rhs_elem;
58           j.data <- j.data + 1;
59           k.data <- k.data + 1;
60         end
61     done;
62     if i.data < n1 then
63       for t = i.data to (n1 - 1) do
64         let lhs_elem = Array.get lhs t in
```

```
65          Array.set a k.data lhs_elem;
66          k.data <- k.data + 1;
67        done
68      else if j.data < n2 then
69        for t = i.data to (n2 - 1) do
70          let rhs_elem = Array.get rhs t in
71          Array.set a k.data rhs_elem;
72          k.data <- k.data + 1;
73        done
74
75  (* Merge sort with mutable datastructures *)
76  let rec mergesort l i_start i_end  =
77    match (i_end - i_start + 1)  with
78    | 0 -> ()
79    | 1 -> ()
80    | _ as len ->
81      if len > !sort_limit then begin
82        let i_mid = (i_start+i_end)/2 in
83          mergesort l i_start i_mid;
84          mergesort l (i_mid+1) i_end;
85          merge l i_start i_mid i_end
86      end
87      else begin
88        let new_array = Array.sub l i_start len in
89          Array.sort (fun a -> fun b -> if a > b then 1 else if a =
      b then 0 else -1) new_array;
90          Array.blit new_array 0 l i_start len
91      end
92
93  (* Merge sort with LWT *)
94  let rec lwt_mergesort l i_start i_end  =
95    match (i_end - i_start + 1)  with
96    | 0 -> Lwt.return ()
97    | 1 -> Lwt.return ()
98    | _ as len ->
99      if len > !sort_limit then begin
100       let i_mid = (i_start+i_end)/2 in
101         Lwt.bind (lwt_mergesort l i_start i_mid) (fun () ->
102           Lwt.bind (lwt_mergesort l (i_mid+1) i_end) (fun () ->
103             Lwt.return (merge l i_start i_mid i_end)))
104     end
105     else begin
106       let new_array = Array.sub l i_start len in
107         Array.sort (fun a -> fun b -> if a > b then 1 else if a =
      b then 0 else -1) new_array;
108         Array.blit new_array 0 l i_start len;
109         Lwt.return ()
```

```
110     end
111
112 (* Asynchronous mergesort with LWT *)
113 let rec async_lwt_mergesort l i_start i_end  =
114   match (i_end - i_start + 1)  with
115   | 0 -> Lwt.return ()
116   | 1 -> Lwt.return ()
117   | _ as len ->
118     if len > !sort_limit then begin
119       let i_mid = (i_start+i_end)/2 in
120         let t1 = Lwt.bind (Lwt_main.yield ()) (fun () ->
      async_lwt_mergesort l i_start i_mid) in
121         let t2 = Lwt.bind (Lwt_main.yield ()) (fun () ->
      async_lwt_mergesort l (i_mid+1) i_end) in
122           Lwt.bind (Lwt.join [t1;t2]) (fun () -> Lwt.return (
      merge l i_start i_mid i_end))
123     end else begin
124       let new_array = Array.sub l i_start len in
125         Array.sort (fun a -> fun b -> if a > b then 1 else if a =
      b then 0 else -1) new_array;
126         Array.blit new_array 0 l i_start len;
127         Lwt.return ()
128     end
```

B.3 Worker Thread

The following code runs on each Java worker thread and processes lightweight threads until interrupted, this loop is relatively small in order to try and keep the bulk of the processing in the lightweight thread rather than the thread management overheads.

Listing B.3: Worker Thread code

```
1  (* Function needs to lock the yielded list, pick up a sleeping
       thread and call 'wakeup' on it,
2      then rinse and repeat *)
3  let run_thread () =
4      while not (Thread.interrupted ()) do
5          Lwt_sequence.lock yielded;
6          let t = Lwt_sequence.take_first_l (fun t -> Lwt.
       try_lock_wakener t) yielded in
7            Lwt_sequence.unlock yielded;
8            match t with
9             | None -> () (* just spin round and try it again *)
10            | Some wakener ->
```

```
11                wakeup wakener ();
12                Lwt.unlock_wakener wakener
13      done
```

Appendix C

Project Proposal

C.1 Introduction

The concept of functional programming is becoming far more widespread and important in recent times. Object orientated and imperative languages like C++ and Java are beginning to use functional concepts such as type inference, first-class functions and more. The C++11 specification defines a new 'auto' type which causes the compiler to infer the variable type from its content[6]. Furthermore, Java 1.8 now includes Lambda Expressions in conjunction with a range of modifications to the libraries that allow its use[8]. Companies such as Jane Street – a quantitative trading firm – use OCaml entirely for all their high performance trading software and algorithms[1], and contribute to the advancement of the OCaml language.

OCaml is a great example of a powerful and robust functional language. It includes everything you would expect from a functional language and more such as object orientated and imperative programming paradigms, and is derived from the highly expressive ML programming language. Its powerful type system provides great type safety, eliminating many of the associated runtime errors; automatic memory management through it's incremental garbage collector and strict evaluation branching from it's ML roots in theorem proving. Parallelism in OCaml is however limited due to a global runtime lock present in the system. The purpose of this lock is to prevent unsafe use of non re-entrant code within the core OCaml library, but more importantly is present due to the fact that the OCaml garbage collector is not multi-core friendly[4]. The resulting issue is that only one piece of OCaml code may be executed at a time leading to very little capacity for parallelism.

Java on the other hand is a widely supported, widely available software plat-

form and programming language used in devices like desktop/laptop PC's, mobile phones, ATM's and even credit/debit cards[9]. Its huge community of developers, wide selection of libraries and ability to write easily portable and distributable software means it has continued to grow rapidly over the years. Java also has a thread safe concurrency library which works in conjunction with its parallel and concurrent G1 garbage collector.

Both languages have their pros and cons however the OCaml-Java project aims to bring the best of both languages together by compiling OCaml to Java bytecode, taking advantage of OCaml's powerful typing system and Java's feature rich library. In addition, OCaml code compiled with OCaml-Java uses Java's G1 garbage collector meaning that the software is able to run in parallel across multiple cores without restriction. OCaml-Java comes with a relatively low-level concurrency library which allows developers to utilise its multi-core capabilities, however these low-level modules can be cumbersome to use frequently, especially considering there are some very popular and powerful concurrency libraries for OCaml – such as Ocsigen's Light Weight Threads (LWT).

C.2 Substance and Structure of the Project

The aim of this project is to port the LWT concurrency library to use OCaml-Java's low-level threading library therefore taking advantage of Java's multi-core processing capabilities and providing an easier, more robust means of writing parallelised OCaml code to run on the JVM. LWT is a widely used monadic concurrency library for OCaml with strong emphasis on the 'light' aspect of their threads. Spawning a thread in LWT is a very cheap operation and as such, threading with LWT fits well with the short-lived-data aspect that is well acquainted with functional languages. Furthermore the results of the project will provide some interesting results into the increased scalability of OCaml code running in parallel across multiple cores.
Thinking modularly, the project can be divided into three distinct sections:

Porting the LWT Library: Porting the library to use OCaml-Java will allow the execution of software written in LWT under OCaml-Java, running on a single thread. This will mainly consist of writing a back-end to act as a pipeline between LWT and OCaml-Java.

Making LWT parallel: Making the ported LWT library parallel will mean utilising many Java threads to execute the LWT threads. These LWT threads are as mentioned very lightweight and are frequently spawned, Java threads

on the other hand are not lightweight and as such a one-to-one mapping of LWT to Java threads will not be possible. To resolve this issue, and thus connect the two sides of the equation, an appropriate scheduler must be written to distribute LWT threads between the fewer Java threads that are active on each core.

Benchmarking: The final aspect of the project is to evaluate how the scalability of software written in LWT scales when run under my ported version as opposed to when run under default OCaml. It will be necessary to convert existing parallel processing benchmarks to use the ported LWT library and also test the scalability on hardware of varying parallel processing power, such as on a dual-core, quad-core and 48-core machine.

C.3 Success Criteria

For the project to be a success I have set the following requirements:

1. I am able to show the scalability difference between two programs using the same piece of LWT code, one compiled using the original LWT libraries under the normal OCaml compiler and one compiled using my ported LWT libraries under OCaml-Java, running on the JVM.

2. Parallel processing benchmark can be used to determine the scalability using LWT under OCaml-Java and using LWT under the normal OCaml environment.

C.4 Starting Point

My experience in functional programming extends as far as the ML courses in Part IA, however I enjoyed completing Project Euler challenges in ML (of which I was normally one of the few people to do so!). Neither have I developed code of a substantial size in ML or OCaml, therefore the best place place to begin would be looking into the structure of open source OCaml code and get a feel for how things are laid out in OCaml. In addition the book 'Real World OCaml'[14] has a public beta available before publication (lucky for me) which will also provide a solid foundation in producing real world OCaml code.

This project relies on the OCaml-Java project, an OCaml to Java bytecode compiler, developed by Xavier Clerc at INRIA. This project provides the ability to use the numerous Java libraries as well as Java's already mentioned parallel

processing capabilities. Furthermore the JVM can be found installed on many different types of hardware and systems meaning the target hardware for OCaml code will be greatly increased by using OCaml-Java. Xavier Clerc, having strong connections with the OCaml Labs here in Cambridge, is also keen to support this project. The version of OCaml-Java I will be using is an early preview release of version 2.0.

Finally getting to grips with the innards of LWT will be most easily achievable by experimenting with the software first hand and diving into the source code available on Github.

C.5 Optional Extensions

Some ideas for optional extensions include:

- Experiment with different schedulers to see which perform better and investigate why.
- Attempt to get higher compatibility rate than initially decided which would mean ironing out incompatibilities with current software written with LWT when run with my ported libraries.

C.6 Timetable and Milestones

17^{th} Oct – 6^{th} Nov Further research into LWT's source. Research OCaml coding techniques.
Milestone: Complete LWT coding tutorial; Read through 'Real World OCaml'; Implement some parallel programs in OCaml-Java 2.0; ready to begin planning.

7^{th} Nov – 27^{th} Nov Continued research into LWT's source. Plan implementation of basic functionality of the ported library.
Milestone: Ready to begin development.

28^{th} Nov – 18^{th} Dec Begin porting LWT to single threaded interfacing with OCaml-Java.
Milestone: Some basic functionality implemented.

19^{th} Dec – 8^{th} Jan Continued porting of LWT to single threaded interfacing with OCaml-Java.
Milestone: Porting complete. Should be able to walk through the LWT tutorial without much/any hassle.

9^{th} Jan – 29^{th} Jan Begin creating the scheduler to manage the distribution of LWT threads to OCaml-Java threads.
Milestone: Fixed number of Java threads are able to be scheduled to, regardless of number of cores on system.

30^{th} Jan – 19^{th} Feb Continued scheduler development.
Milestone: Number of Java threads scales with the number of cores present on the system.

20^{th} Feb – 12^{th} Mar Begin conversion of parallel processing benchmarks.
Milestone: Most of benchmarking software conversion complete, fixed relevant bugs which may arise from testing with the benchmark.

13^{th} Mar – 2^{nd} April Testing with benchmarks, finishing dissertation evaluation and write-up.
Milestone: Relevant scalability graph data accumulated.

3^{rd} April – 23^{rd} April Finalising dissertation write-up.
Milestone: Dissertation complete!

C.7 Resources Required and Backup

A list of required resources:

- My own Machines:
  - Macbook Air 13" 2013 (8GB RAM, 512GB Storage, OSX/Windows 8)
  - Desktop Computer (Ubuntu 13.04/Windows 7, Intel Core2 Quad, 5TB Storage)
- Machines in College
- A many-core machine in the SRG (32 - 48 Cores; e.g. 'Roo')
- OCaml-Java and Sources[1]
- OCaml[2]
- LWT Sources[3]

[1]OCaml-Java: http://ocamljava.x9c.fr/
[2]OCaml: http://ocaml.org
[3]LWT: http://ocsigen.org/lwt/

Backups will be provided by Dropbox, Skydrive, my own personal USB stick and my personal storage on the PWF facilities. Cloud backup will be incredibly useful for keeping my data accessible from many locations and also avoiding local corruption which could occur at any time. Version control (and also the main copy of my dissertation/project) will be provided by Github. I chose Github since I am already very familiar with git version control; Github provides cloud storage so I'm able to retrieve and work on my files from various locations; furthermore Github provides free private repositories for students.