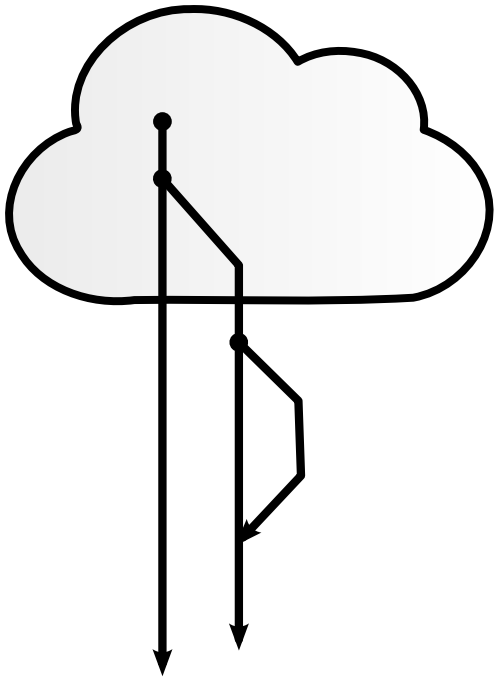
Sebastian Hollington

Cirrus

Distributing Application Threads on the Cloud



Computer Science Tripos, Part II

St. John's College

16th May, 2011

The word 'Cirrus' is used to describe the "thin, wisp-like" high-altitude clouds that can form almost any shape, and cover vast areas of the sky, typically indicating fine weather. The Cirrus cloud acts as a perfect metaphor for the intentions of this project: aiming to offer a light-weight and efficient high-level abstraction of a cloud which leaves a positive impression on those working beneath it.

Proforma

Name:	Sebastian Hollington
College:	St. John's College
Project Title:	Cirrus – Distributing Application Threads on the Cloud
Examination:	Computer Science Tripos, Part II, June 2011
Word Count:	11,986
Project Originator:	Sebastian Hollington
Supervisors:	Malte Schwarzkopf and Derek G. Murray
Overseers:	Dr. Timothy G. Griffin and Dr. Markus Kuhn

Original Aims of the Project

The aim of this project was to implement a framework that could utilise the power and the scalability of distributed computing using concepts familiar to the traditional single-system programmer. The finished product should hide the technicalities and nuances of cloud computing by exposing a light-weight and intuitive threading-like API. This API together with other complementary abstractions should be able to successfully shorten the execution time of a variety of different algorithms by distributing them over a typical cloud computing cluster.

Work Completed

The finished product consist of a self-contained C library that distributes "cloud threads" over a CIEL distributed computing cluster. Cirrus is shown to successfully distribute an entire suite of test cases and reduces the running time for computationally intensive algorithms by seamlessly taking advantage of all available worker nodes on the cloud. Several extensions were implemented that either enhance the flexibility, optimise the execution, or further integrate the Cirrus framework with the cloud, making it suitable for a far larger range of algorithms than originally expected.

Special Difficulties

None.

Declaration

I, Sebastian Hollington 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:

Date: 16th May, 2011

Acknowledgements

This project would not have been possible without the encouragement and enthusiasm of the following individuals:

- **Malte Schwarzkopf**, for invaluable guidance from the very beginning, making sure that I stuck to my milestones, and for the countless contributions shared on topics that furthered this ambitious project.
- **Derek G. Murray**, for his relentless stream of new ideas, and for giving up his time to provide unrivalled insight into how CIEL works and how its features could be best exploited.

Contents

Chapter 1

Introduction

1.1 Motivation

With chip manufacturers signalling an end to the Gigahertz race, software engineers have recently been forced into parallelising their algorithms to achieve the best performance on multi-core CPUs. Since then, we have seen the cloud computing paradigm take off with commercial offerings such as Amazon EC2[1] making infrastructure-as-a-service available to anyone with a credit card.

While lots of research has gone into developing schedulers and execution engines that can manage a cloud cluster, uptake of cloud-computing has been limited by the difficulty of migrating code from thread to cloud-based scheduling. This could potentially involve baking the cloud-paradigm and infrastructure into the application or having to rewrite code in new bespoke scripting languages, following strict scheduling constraints. Today, users have access to thousands of cores across thousands of computers and yet their thread-parallelised programs can only exploit the computing power from the cores available locally.

Many modern-day applications that implement parallel algorithms use a threading API to leverage multiple cores on the local machine (most commonly POSIX.1c [10] for C/C++). If there were a similar API that would instead distribute tasks on a cloud instead of confining execution to the local computer, parallelisable algorithms could be easily ported to take advantage of thousands of cores from the cloud.

Such a framework would be flexible enough to distribute many different types of programs, particularly those requiring large amounts of computation, by programmers who may only be familiar with traditional system threading. The framework's development-time saving aspect could make its adoption appeal in commercial environments where large data sets have to be processed in batches, but mathematical and scientific computation could potentially be an even better use-case. Finally, the computing power unlocked by such a frame-

[1] http://aws.amazon.com/ec2/

work could even benefit end-users if their applications allow them to offload their work, e.g. home video encoding, to consumer cloud services.

1.2 Related Work

A lot of research has gone into distributing particular programming models, for example Google's MapReduce [3], but these models are of course more restrictive than using a generic multi-threading programming model.

Alternatively, the Piccolo [17] distributed computing platform is designed specifically to make coding distributed applications easier for the traditional single-system programmer. However, the framework is again quite restrictive in how programs are executed and it does not have the ability to scale transparently at runtime to all available cloud nodes. This mitigates a lot of the advantages of having elastically allocated cloud services.

Alternatively, Douceur *et al.* developed UCop (the "utility coprocessor") – a system...

> "...that makes it cheap and easy to achieve dramatic speedups of parallelizable, CPU-bound desktop application using utility computing clusters in the cloud." [4]

UCop does not however attempt to distribute tasks at thread granularity, and still requires major architectural changes to the software design to move a program from a multi-threaded, single-system environment to running across a cloud network.

A significant amount of research has also gone into *distributed shared memory* (DSM) implementations which attempt to provide more heavy-weight levels of abstraction for distributed computing environments. Kerrighed, OpenSSI and OpenMOSIX [13] are DSM solutions that set out to present a distributed computing cluster as a *single system image* (SSI) where, for example, processes may be automatically moved to remote idle workers if competing for CPU time on a single machine.

These systems can however be quite difficult to initially set-up, and while they are quite effective at load-balancing, they experience large overheads from keeping all memory in a consistent state across networks. SSIs are also intrinsically handicapped simply because they typically distribute at process-level granularity. Lazy developers might not consider even for a moment how they could better partition or componentise their distributable algorithms which can lead to needlessly inefficient executions.

In conclusion, I have been unable to find a flexible and light-weight solution that is able to ameliorate the work required to distribute *within* an application, while still harnessing the full potential of the cloud in a fully automated and scalable fashion. Cirrus intends to do exactly this by meshing a universally interfacable thread-like distribution API with friendly abstractions of the distributed computing paradigm.

Chapter 2

Preparation

In this chapter, I outline the core functionality in a standardised threading library and use this information to draw up a list of requirements for Cirrus. Using this list, a structure for the project is then proposed. I then go on to investigate the cloud frameworks that would be most suitable for implementing those requirements.

The latter part of this chapter will then detail the tools chosen for implementation, including the programming languages, development environments and the version control and backup systems. Finally, I briefly outline any noteworthy software engineering techniques employed over the duration of the project and summarise the key decisions made.

2.1 Investigating Threads

In order to implement a thread-like interface for cloud scheduling, I first had to familiarise myself with how threads are normally spawned and executed. To understand traditional threading APIs, I decided to look at the POSIX.1c [10] pthread library. Although the full command-set contains over 100 functions, only a minimal subset provide core functionality.

A thread is created using the following function:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr, void
   *(*start_routine)(void *), void *arg);
```

This call creates a new thread and passes out a pointer that can be used to uniquely identify that thread in the future. The entry point of the thread is specified by passing a function pointer in as the `start_routine` argument. When the thread starts, it executes this function, passing it the argument `arg`. The thread then executes until `start_routine` returns a value that is saved and can later be interrogated.

In order to retrieve the value returned by a thread, POSIX defines the following API:

```
int pthread_join(pthread_t thread, void **value_ptr);
```

This function blocks the calling thread's execution until the specified `thread` has finished executing. The return value of `thread` is then passed back to the calling thread using `value_ptr`. If the return value is available immediately (i.e. the thread has already completed) this function returns straight away.

It should also be noted that functions running under a thread are first-class citizens in that they are also able to create and join child threads arbitrarily.

In order to provide a thread-like API, Cirrus must implement an equivalent version of at least these two core functions. The conventions identified here were used to shape the construction of the Cirrus API, outlined later in section 3.2.

Another important aspect of threading that Cirrus must tackle is how to share memory between threads. In a single process environment, traditional system threads share the *same* memory space which mean that all threads can share data simply by reading/writing to prearranged locations. However, as Cirrus will not only be executing threads in a different process but will be executing that process on a different computer, Cirrus must provide an alternate fast and efficient means for cloud threads to explicity share data over a network.

2.2 Requirements Analysis

2.2.1 Behavioural Requirements

Following my investigation, a framework that spawns "cloud threads" must meet the following requirements:

Requirement 1
The framework must provide a way to execute a function on an arbitrary node of a distributed cluster. This should be done with the abstraction of a "cloud thread" which uses the function as its entry-point.

Requirement 2
The framework must allow the aforementioned entry-point to accept input from the caller (e.g. through at least one argument).

Requirement 3
The framework must be able to block and wait for a specific cloud thread's return value.

Requirement 4
The framework must allow threads themselves to schedule and block on the output of other threads both arbitrarily and independently.

Requirement 5
The framework must provide a means for threads to share data between themselves.

2.2.2 Structural Requirements

In keeping with the traditional threading library that was investigated in section 2.1, it was clear that Cirrus should implement its API as a consolidated library (hereafter referred to as `libCloudThreads`).

It was realised that the other key component of Cirrus is the interface with the distributed computing cluster (i.e. the cloud). This functionality is typically the responsibility of a *distributed execution engine*.

Distributed execution engines (DEEs) are used to abstract away the complexity of distributed computing. It is this service that manages and distributes tasks to the available nodes on a cloud computing cluster. While it would have been of course possible to write a custom DEE for Cirrus from scratch, a lot of research has already gone into this area so I believed it was preferable to extend or adapt one of the tried-and-tested solutions. A selection of DEEs are investigated in section 2.3.

2.3 Investigating Distributed Execution Engines

As explained in section 2.2.2, Cirrus requires a DEE to manage and distribute tasks on the cloud.

This project was inspired by the work the Systems Research Group are doing on their fully dynamic DEE, CIEL. However, for the purposes of completeness, I felt it necessary to look into the alternatives to make sure that CIEL was in fact the best option for Cirrus.

Google MapReduce [2] 2004

A patented software framework consisting of two fixed functional stages: a set of `map` operations followed by a `reduce` stage. This enforces a static two-stage data-flow and is therefore used mostly for processing large data sets. MapReduce inspired the free open-source Apache implementation, Hadoop[1].

Microsoft Research's Dryad [11] 2007

Instead of having a two-stage dependency chain, Dryad can accommodate any static directed acyclic graph (DAG). However, similar to Google MapReduce, control flow cannot be altered during execution.

[1] `http://hadoop.apache.org/`

Google Pregel [14] 2009

Named after the infamous river that introduced the seven bridges in Königsberg, Pregel improves on Dryad and MapReduce by employing an iterative implementation of Valiant's Bulk Synchronous Parallel (BSP) model [18]. Tasks are executed concurrently, but periodically pause at a determined interval to see whether any required results have been produced.

Iterative Google MapReduce [6] 2010

By applying the `map` and `reduce` stages iteratively, MapReduce is able to support dynamic data flows but at the expense of losing fault tolerance if any nodes fail or become unreachable whilst processing a job.

Piccolo [17] 2010

Piccolo uses shared tables and checkpointing to provide a distribution framework that programmers can use to develop distributed programs. However, the engine does not allow applications to scale transparently to an arbitrary number of cloud nodes and task dependencies must still be determined at compile time.

CIEL [15, 16] 2010

A lazy-evaluative distributed execution engine developed by Murray *et al.*. It is designed to support fully dynamic task dependencies and data flow with transparent scaling. Its deterministic scheduling facilitates strong fault tolerance and robust error recovery and offers extendable front-ends for different types of task execution.

Feature	MapReduce	Dryad	Pregel	Iterative MR	Piccolo	CIEL
Dynamic Flow Control	✗	✗	✓	✓	✓	✓
Task Dependencies	Fixed (2-stage)	Fixed (DAG)	Fixed (BSP)	Fixed (2-stage)	Fixed (1-stage)	Dynamic
Fault Tolerance	Transparent	Transparent	Transparent	✗	Checkpoint	Transparent
Data Locality	✓	✓	✓	✓	✓	✓
Transparent Scaling	✓	✓	✓	✓	✗	✓

Table 2.1: Analysis of features provided by existing distributed execution engines from [16].

The features present in each of the DEEs are summarised in table 2.1. With the exception of CIEL, all of the data flows (as shown in figure 2.1) are static i.e. they cannot change during the execution of a job. To properly mimic the traditional thread scheduling investigated in section 2.1, Cirrus requires *dynamic* dependencies and data flow which left CIEL as the only execution engine suitable for this project. Cirrus programs can also exploit CIEL's transparent scaling[2].

This choice also meant that Cirrus benefitted from being supervised by those who designed and implemented the engine it extends.

[2]Where the number of worker nodes does not have to be specified for a job at compilation time.

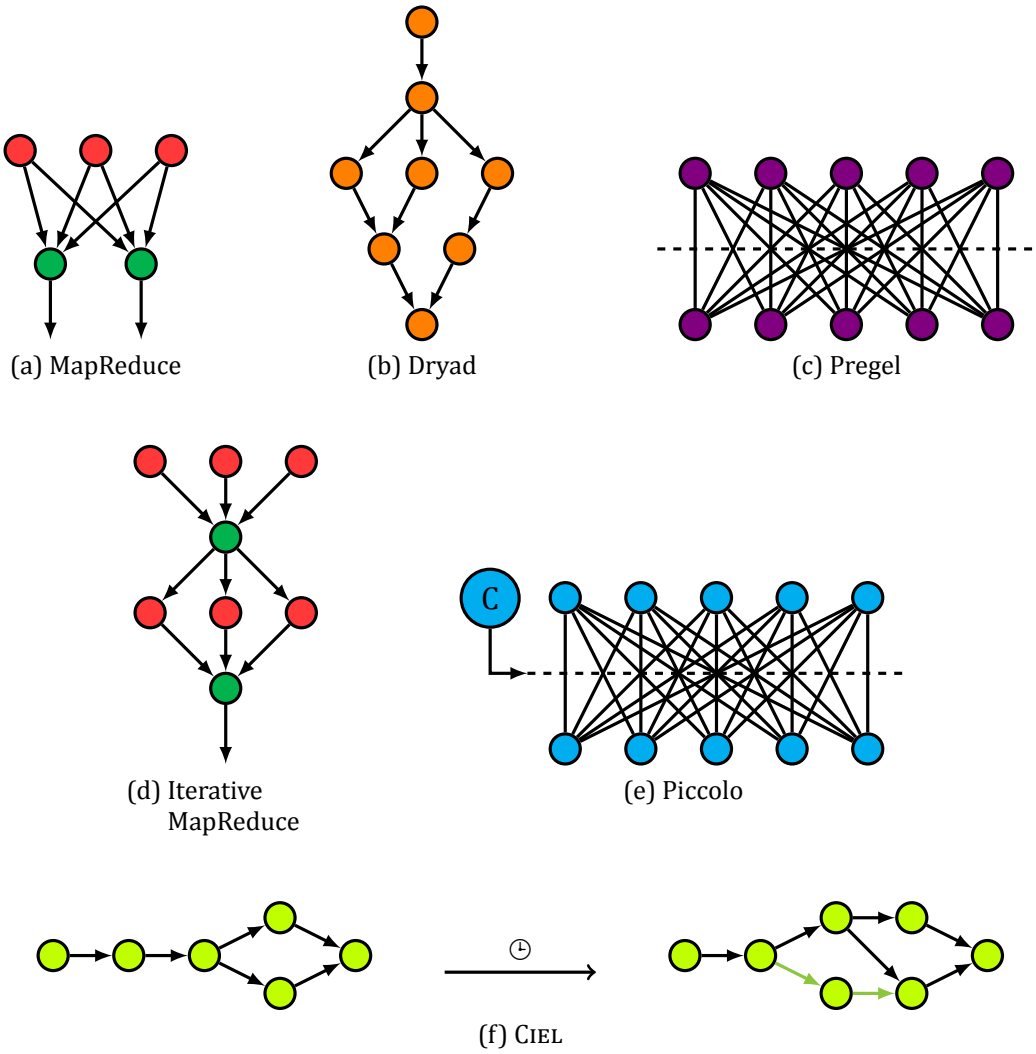


Figure 2.1: Graphical representation of the type of data flow that each DEE supports.

2.4 Further Implementation Decisions

Requirement 1 states the need to execute a process's function on a *remote* worker node. This will require halting a process and serialising a snapshot of it so that it can be resumed later at another location. This technique is often referred to as *process checkpointing* but is more commonly used for recovering long-running processes upon system crashes.

The *process checkpointing system* that I chose for this project was the Berkley Labs Checkpoint Restart project [8] (hereafter referred to simply as BLCR). BLCR provides a C library that interfaces with a Linux kernel module[3] to perform checkpointing operations. This of course restricts the execution environment for `libCloudThreads` to the Linux OS however the operating system is freely available, and most cloud providers offer Linux-based nodes as the default choice.

2.5 Choice of Tools

2.5.1 Programming Languages

Cirrus's `libCloudThreads` should be able to target the largest audience possible and because of this, I decided to write it in C as most higher level programming languages offer bindings to the standard C calling convention. To take advantage of more flexible language syntax, the C99 [12] standard was used with the GCC GNU extensions.

`Doxygen` [19] style comments (similar to JavaDoc but for C/C++) were used to document the library API inline in the source code. When the project is built, `Doxygen` generates both HTML and LaTeXfiles documenting the library (see appendix B). Another reason for using `Doxygen` is that most C/C++ IDEs are able to parse the comments and present them as visual aids to the programmer.

In order to extend CIEL so that it could correctly handle and schedule Cirrus tasks, I had to also learn Python which the CIEL runtime is implemented in.

2.5.2 Development Environment

The use of BLCR restricts the execution (and therefore the development) environment to a Linux kernel no newer than 2.6.34. With those constraints in mind, I decided to opt for the familiar and widely used Ubuntu 10.04 LTS (Lucid Lynx) distribution as the chosen development OS.

[3] After corresponding with the BLCR researchers, I determined that the latest kernel officially supported by BLCR is 2.6.34, which is required for the kernel module to function correctly.

All of the implementation was carried out in a Parallels 6[4] virtual machine on a Mac OS X 10.6 machine. For testing purposes, two concurrent CIEL worker processes were run to simulate a cloud network environment locally (see section 4.2.1).

The University's Personal Workstation Facility (PWF) was used for backing-up project documents and also hosting one of the version control repositories (see below).

Integrated Development Environments

Code::Blocks[5] 10.05 was used for C-based development which included the writing of `libCloudThreads`, the C profiling module, and the suite of test programs.

Eclipse[6] Galileo and the pyDev[7] plug-in were used for any Python-based work done in extending the CIEL framework.

Version Control

Two separate version control repositories were used extensively throughout this project.

Work Repository `svn+ssh://sh583@linux.pwf.cam.ac.uk/~/...`

This private subversion (SVN) repository was hosted in my user space on the University's PWF server. It was used to back-up and keep track of the LaTeX source code used to generate this dissertation and the earlier project proposal.

Code Repository `git://github.com/sebholl/ciel.git`

This publicly available Git repository is hosted on GitHub.com and was forked from the main CIEL repository **mrry/ciel**[8]. It is the development branch where all code written for this project was committed to.

Whilst I have been using subversion on a regular basis over the last few years, this project required me to learn how to use the Git version control system. I did however find Git to be similar to the Mercurial collaborative version control system that was used in my Part IB group project.

[4] http://www.parallels.com/
[5] http://www.codeblocks.org/
[6] http://www.eclipse.org/
[7] http://pydev.org/
[8] git://github.com/mrry/ciel.git

Backup Strategy

Both of the version control repositories were located on corporate or commercial servers that have rigorous back-up policies, however in addition to these central repositories, working copies were regularly pulled from the servers to my personal notebook computers in case the servers went down. Files on my local machine are backed up to rewritable Blu-Ray discs on a monthly basis and Cirrus data was included with that.

2.6 Software Engineering Techniques

For the most part, an iterative *Agile Development* [9] strategy was adopted throughout this project. Typically, a specific test or use-case was constructed following a meeting with supervisors and then the library was extended accordingly to enable the test to run. Code changes were committed to the repository whenever a section of work was completed and when appropriate, references to the corresponding test-cases were mentioned in the accompanying commit log.

Whilst most of the library was written in the low-level, procedural C, appropriate code-writing practices – such as the use of scope modifiers – were used to hide internal functions from other parts of the code. External libraries were used for common functionality and data structures, eliminating the time required to re-implement and test data structures that in other languages would normally be provided by the standard libraries.

As mentioned in section 2.5.1, documentation was added inline in the code to keep it close to the corresponding function declarations, promoting consistency and improved maintainability. Object-orientation was used when extending the Python-based CIEL framework to improve structural clarity and also in keeping with the rest of the CIEL code base.

2.7 Summary

In this chapter, I presented the research done investigating the POSIX.1c threading library. Using this information, I drew up the requirements of the project and from this I outlined a structure for Cirrus. The use of the CIEL distributed execution engine was justified, as was the need for a third-party process checkpointing system (BLCR). I also decided Cirrus's API should be packaged as a universally accessible C library named `libCloudThreads`.

I then justified the use of a Ubuntu 10.04 development environment, along with Subversion and GitHub version control repositories. Finally, in section 2.6, I declared *Agile Development* as the software development model that should be used.

Chapter 3 follows with extensive details as to how the project was actually implemented; the design decisions made; and how various problems were tackled.

Chapter 3

Implementation

In chapter 2, I outlined the core structural requirements of Cirrus. The three core components are as follows:

1. **`libCloudThreads` API**

   This API is the front-end for Cirrus and is the interface application developers should use to distribute their threads on the cloud. This component implements the key ideas Cirrus introduces to map would-be system threads to cloud tasks that can be distributed. The core functions mimic the traditional threading commands investigated in section 2.1 as much as feasibly possible.

2. **Process Checkpointing Interface**

   This `libCloudThreads` component is responsible for interfacing with the BLCR library (described in section 2.4): packaging the current process such that it can resume on a remote worker node. This component must be able to produce an executable checkpoint file that can be distributed on the cloud (see below).

3. **Cloud Framework Interface**

   This third component is responsible for *all* communication and interaction with the CIEL cluster. This entails dispatching HTTP-based instructions to the master node; translating `libCloudThreads` data into CIEL -interpretable values; and accessing/retrieving data distributed over the cloud network.

   Due to its nature, this component is actually split into two halves: a C-based runtime as part of `libCloudThreads`; and a set of Python-based *task-executor*s that are used by CIEL worker processes (see section 3.1).

Figure 3.1 shows how these components interact to form Cirrus. In this chapter, I will first give a brief overview of how the CIEL DEE distributes tasks on a cloud; and then go on to address how each of Cirrus's requirements were implemented and exposed as an API.

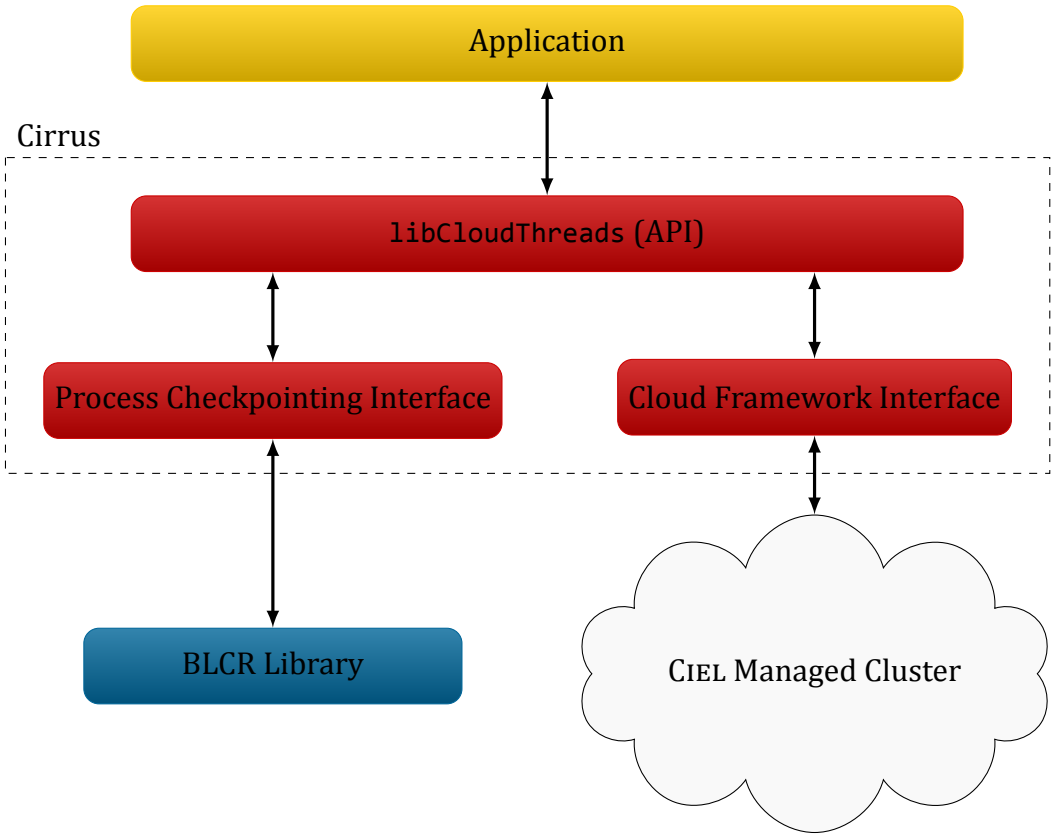


Figure 3.1: Structural overview of Cirrus.

3.1 Cloud Overview

A CIEL environment consists of a master node and a set of worker nodes. The role of the master is to accept, queue and assign tasks to any registered workers, as shown in figure 3.2. Both master and worker nodes are controlled using HTTP requests.

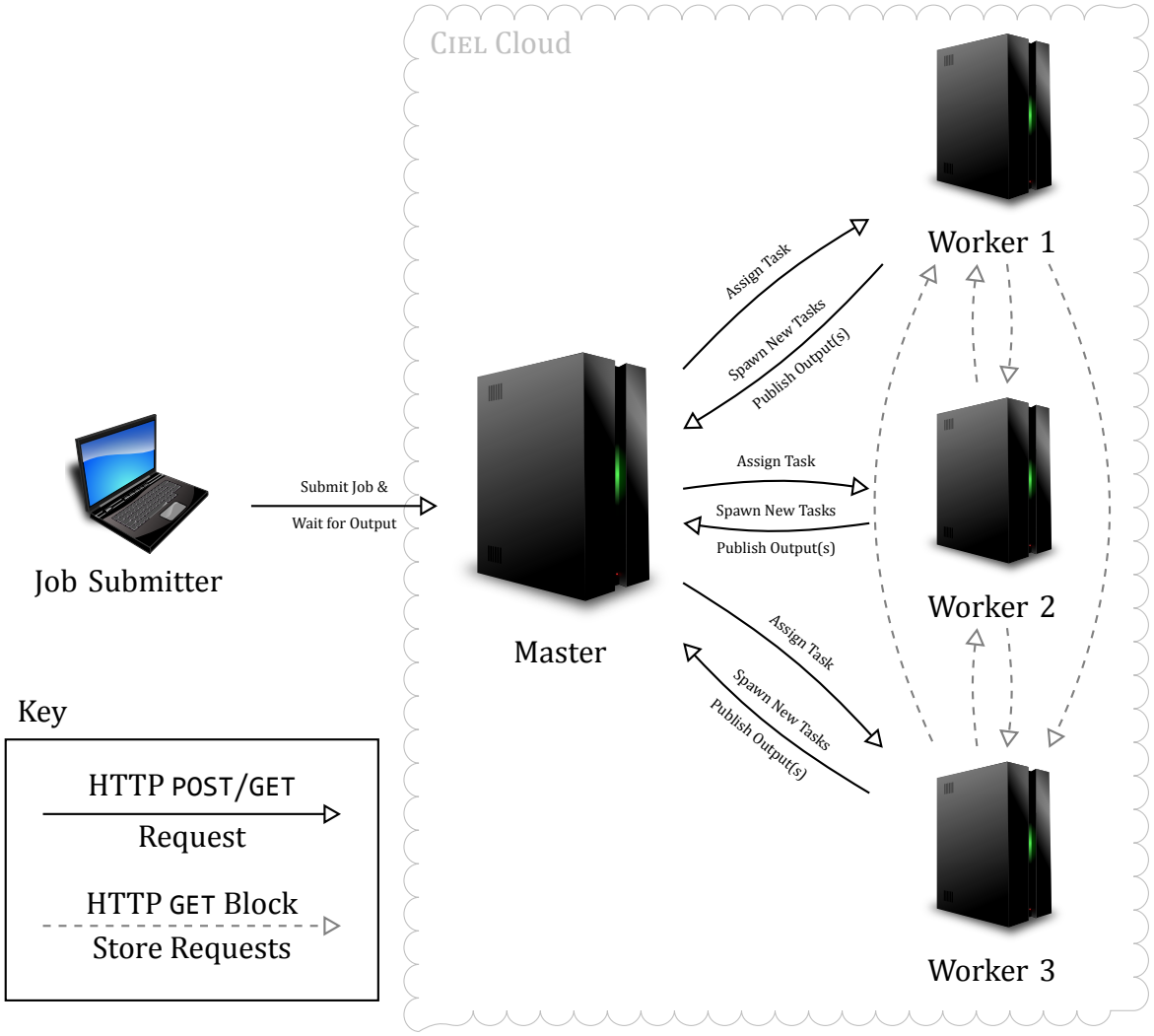


Figure 3.2: A typical CIEL *network.*

A CIEL task has a descriptor that contains any information that the master needs to schedule/assign the task (see section 3.2), but also any information that a worker needs to run it. Each worker has a local block-store that tasks can publish data to so that they can be retrieved by other nodes.

While each task has its own unique task ID, the most important values that determine the scheduling of a task is its output ID and the set of IDs that represent its dependencies. Lazy

evaluation is used to recursively evaluate only the tasks necessary to produce a value for a specific output ID – these tasks are referred to as "critical tasks".

Each task must also specify a *task executor* (see section 3.2.2) which is responsible for carrying out a task's work. When a task is scheduled for execution, it is assigned to the first available worker that advertised support for the task's executor. Once a task is assigned and sent to a worker, the task executor is responsible for carrying out the task using any additional arguments supplied in the task descriptor. The worker must then advertise the availability of any task outputs upon completion.

If a critical task is blocking on an ID that has just been published, the master then marks the task as runnable and attempts to find a free worker that supports its prescribed task executor. When this task is assigned, the location of its dependencies are passed onto the worker so that they can be retrieved and made available in the local block store for when the task runs.

3.2 Spawning Cloud Threads

If a task is the vector used to carry out a portion of work on a cloud network then, to distribute threads, a mapping was needed to encapsulate a thread as a task.

I implemented the construction of a thread task as follows:

The user calls `cldthread_create( fptr, arg )`, where `fptr` is a function-pointer to the thread's entry point, and `arg` is the value of the single argument that is passed to it. At this point, a checkpoint of the process is taken using `blcr_fork()` (see section 3.2.1). The resulting checkpoint file is saved to the current worker's local block store and a new Cirrus task is spawned that, when scheduled, will resume this process on an arbitrary node. The unique output ID of the new thread task is then returned to the user.

When the process is resumed, the return value of `blcr_fork()` is used to determine whether this process is executing as a new thread task. If so, it will then execute `fptr( arg )` and when it returns, the thread output will be saved to the local block-store, published and the process will be terminated. This behaviour is summarised in Listing 3.1.

Tasks are spawned by `POST`ing a JSON-encoded task descriptor to a master node URL. To enable `libCloudThreads` to spawn tasks itself, information about the CIEL environment (such as the task executing the current process, and the location of the master) must be passed to the Cirrus runtime. This was implemented using custom task executors (see section 3.2.2). `libCloudThreads` then uses this information to generate deterministic task and output IDs for the new task descriptor.

cJSON and cURL libraries are used to encode and subsequently dispatch the spawn request to the master.

```
cldthread_create( fptr, arg ):

  /* Fork the process by creating a checkpoint file */
  forkresult = blcr_fork( tmppath )

  if ( forkresult == 0 ): /* Resumed process checkpoint */

    /* Call thread entry-point and capture the thread result */
    threadresult = fptr( arg )

    /* Save thread result to the local worker's block-store */
    ciel_serialise_to_block_store( threadresult )

    /* Terminate now that the thread has completed */
    exit( EXIT_SUCCESS )

  else if ( forkresult > 0 ): /* Checkpoint succeeded */

    /* Move checkpoint file to worker's block-store */
    chkptfileid = ciel_move_to_block_store( tmppath )

    /* Create a new CIEL id to represent the new thread */
    newthreadid = cielID_create( ... )

    /* Build task-descriptor and submit it to the CIEL master */
    ciel_spawn_task( newthreadid, chkptfileid, ... )

    /* Return the thread ID to the user */
    return newthreadid

  else if ( forkresult < 0 ): /* Checkpoint operation failed */

    print "An error occurred."
    exit( EXIT_FAILURE )
```

Listing 3.1: Pseudocode showing how a cloud thread task is forked from the original process.

I should at this point emphasise that due to the lazy-evaluative nature of CIEL, spawned Cirrus tasks will not be executed until a required scheduled task demands the task's output as a dependency. This nuance is tackled later in section 3.3.

3.2.1 Process Checkpointing Interface

Despite being an integral part of the project, this was a relatively straightforward component to implement compared to the cloud interface. As a reminder, process checkpointing is required to be able to continue a process on a different computer so that the cloud thread's function pointer can be executed (see section 2.4).

All of the implementation details of the chosen checkpointing framework (BLCR) were abstracted away, exposing just two functions to be used by `libCloudThreads`:

```
int blcr_init_framework( void );
```

This trivial function initialises `libcr` (the BLCR C library), outputting any errors relating to the installation of the required kernel module and returns a non-zero value upon success. However, the important command implemented is...

```
int blcr_fork( const char *filepath );
```

...which is a wrapper function I wrote for BLCR that checkpoints a process and writes the checkpoint out to the specified file-path. This checkpoint file can then be sent over a network and invoked using a `cr_restart` command line, at which point the process will carry on where it left off. Ideally, the process should have little idea of what just happened. As such, a method is required to be able to tell apart the process that *invokes* the checkpointing routine, and the process *resulting* from the checkpoint being resumed. To achieve this, `blcr_fork()` is designed to act similarly to the Unix `fork()` command in that its return value reveals the execution context of the returning code.

Return Value	Description
1	Returned to the initiating process to indicate checkpointing success.
0	Returned to a resumed checkpoint process.
-1	Returned to the initiating process if there was an error.

Table 3.1: Return values of `blcr_fork()`.

The cloud thread abstraction part of `libCloudThreads` is responsible for calling `blcr_fork()` with a generated temporary path. It then instructs the cloud-interface component of `libCloudThreads` to atomically move this file into the local block-store and send a message to the master to spawn a new task.

3.2.2 Executing Cloud Threads

The task executor acts as the main bridge between the CIEL worker process and external languages/code. When I forked the CIEL trunk, there were many built-in executors such as `ProcessExecutor`, `JavaExecutor` and `CExecutor`. Unfortunately, these were designed simply to execute an arbitrary program or class file and return. Only the SkyWriting script [15] executor could spawn other tasks and as the requirements of the project dictated that a process running `libCloudThreads` could create threads arbitrarily, I was required to write a new set of executors.

As explained, the Cirrus executor has to collect and pass any information about the cloud environment to the `libCloudThreads` runtime. This information is required for spawning new tasks and committing output from within the `libCloudThreads` library, and includes...

- The current task's ID.
- The current task's output ID.
- The location of the master node (given as a hostname:port pair).
- The location of the current worker node (given as a hostname:port pair).
- The paths of any files that contain the values of the task's dependencies.
- The path of the current worker's block-store.

Using inheritance, the **CielExecutor** base class was extended such that in addition to the existing book-keeping, it collects the information listed above into a data dictionary of name-value pairs. The next step was to determine the best way to pass this information to a Cirrus program, initially when it starts for the first time, but also when the process resumes as a new task. This was achieved by further extending the **CirrusExecutor** class into two separate task executors: **CloudAppExecutor** and **CloudThreadExecutor** (see figure 3.3).

CloudAppExecutor is responsible for tasks that want to start a Cirrus program for the first time. A Cirrus program is started just like any other process, but this executor also passes the cloud information through to `libCloudThreads` by setting environment variables for the new process. CloudApp tasks will most likely be spawned as the root task for a job.

The second executor is responsible for handling the tasks that are spawned *by* Cirrus for running a thread. To execute, **CloudThreadExecutor** resumes a process by calling BLCR's `cr_restart` command-line utility with the task's accompanying checkpoint file. It must then update the environment variables of the resumed process to reflect its new state.

This proved initially difficult as *all* process information (including any environment variables or file descriptors) are serialised at the point of checkpoint and restored *in entirety* when resumed. Any environment variables set when starting `cr_restart` were simply not visible to the resumed process. To solve this problem, a named pipe is created between **CloudThreadExecutor** and the Cirrus process. The executor writes the respective environment variable names and their updated values and these are read in one at a time by `libCloudThreads` immediately after the process resumes. This ensures that all information is up-to-date and consistent as soon as `libCloudThreads` hands back to the program code.

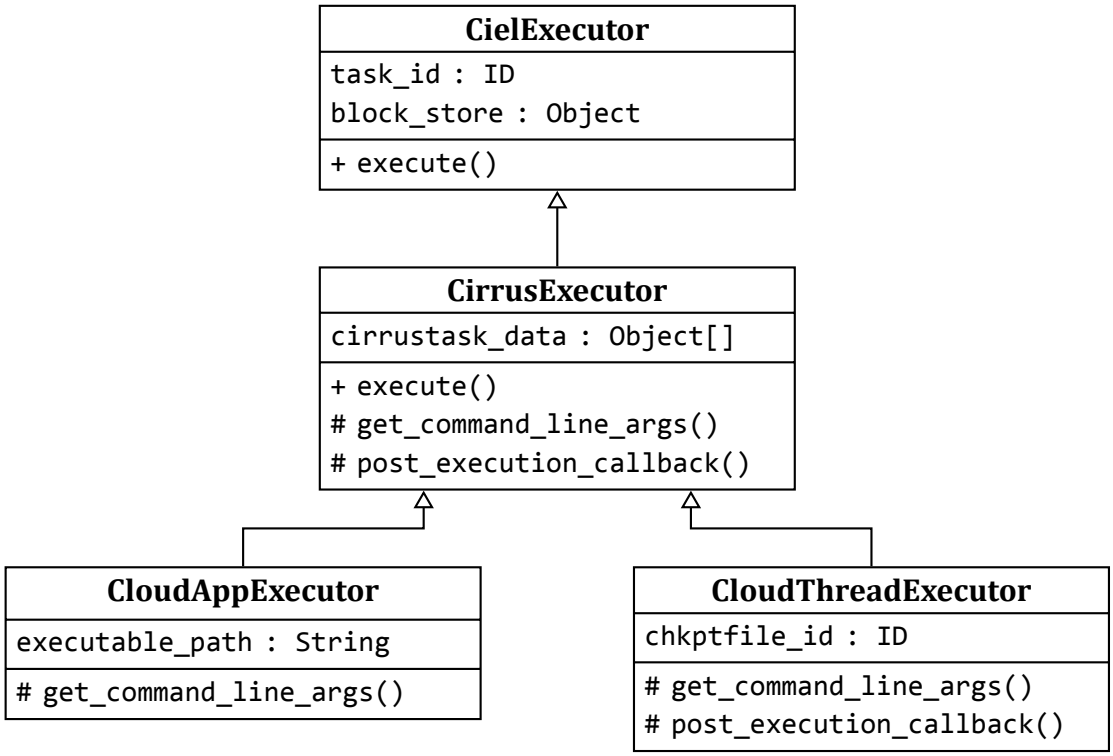


Figure 3.3: Cirrus executor class hierarchy.

3.2.3 Determinism

The CIEL framework relies on task execution and scheduling being *deterministic*. A consequence of this means that a task should be identified not by its own ID, but instead by the output it is proposing to produce. As a result, all Cirrus tasks are referred to internally using simply their output ID.

Output IDs are generated uniquely for each new thread to ensure that it can be scheduled without being shadowed by other tasks. This shadowing phenomenon is however leveraged when implementing the memoisation extension in section 3.8.2. These IDs are then SHA1 [5] hashed to maintain an upper-bound on their length.

3.3 Joining Cloud Threads

The next feature that I had to implement was *thread joining*. Section 2.1 describes the behaviour of a traditional thread join accordingly...

> "This function blocks the calling thread's execution until the specified `thread` has finished executing. The return value of the `thread` is passed back to the calling thread."

As such, to join a *cloud* thread we must force its execution, block until it has completed, and then provide a means for the user to retrieve its return value. This behaviour is exposed using a `cldthread_join()` command.

For a scheduled task to execute in a lazy-evaluation environment, the task's output ID must appear in at least one of the dependencies along a job's critical path. And because of CIEL's requirement for determinism, task dependencies can only be specified at the point where a task is *spawned*.

Therefore to force execution of Cirrus tasks, I was required to introduce the notion of CIEL "continuation tasks". A continuation task is simply a task that, when executed, resumes the work of a previous task. However, by scheduling it as a new task, one is able to change the original task's dependencies.

When `cldthread_join()` is called, `libCloudThreads` first tries to retrieve the supplied IDs from the local block store. If all of the IDs are already available, the function returns straight away, otherwise `libCloudThreads` spawns a continuation task that is dependent on the outputs of the supplied tasks but with the same output ID as the current task. The current task then terminates, dropping the original critical path. Noticing this, the CIEL master will then attempt to evaluate the new critical path provided by the continuation. This results in the thread and the continuation being scheduled and executed as desired (see figure 3.4).

The CIEL task scheduler will block a critical task from running until all of its dependencies are available. This effectively causes the original task to block until the other tasks have completed – matching the desired `pthread_join()` behaviour.

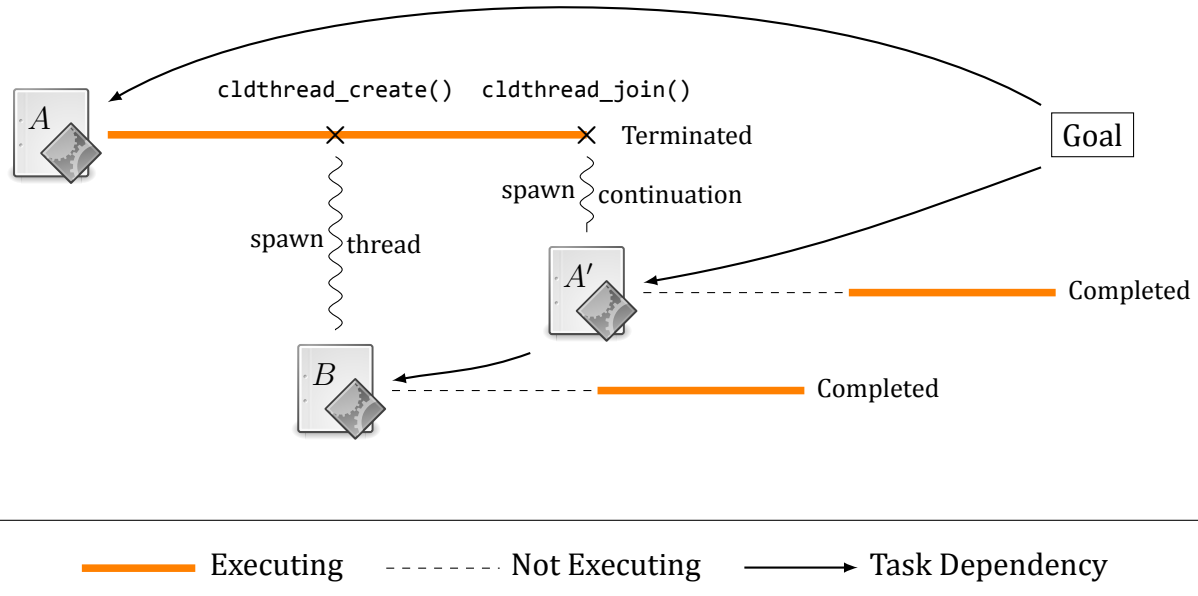


Figure 3.4: Continuation Tasks – task A' is spawned as a continuation task for A to force the evaluation/scheduling of B in the lazy-evaluative CIEL *environment.*

To spawn continuation tasks, I utilised the existing process checkpointing routines implemented in section 3.2 and adapted the internal `libCloudThreads` routines so that a resumed checkpoint can either execute as a thread, or execute as a continuation.

It was also beneficial (from an optimisation perspective) to generalise the Cirrus join implementation so that it can wait on *multiple* output IDs (not just one as prescribed by POSIX threads). Most of the test programs written require waiting for a set of threads to complete and, by exposing this intention to Cirrus, `libCloudThreads` can pass this information onto the CIEL scheduler so that it can allocate tasks more intelligently.

Section 3.4 goes on to describe how Cirrus stores and handles the return values that a developer would want access to following a `cldthread_joins()`.

3.4 Cloud Thread Values

This section is concerned with how Cirrus threads return values and how they are passed around the cloud network.

A major hurdle in integrating C programs into a cloud environment was serialising data so that its type information was preserved. In addition, CIEL tasks can be spawned using the SkyWriting [15] scripting language and so it would be extremely useful if SkyWriting was able to understand a Cirrus program's result.

To be able to pass values between tasks and workers, `libCloudThreads` needed a way to save and query type information alongside data. In order to do this, I defined a new C `struct` called `cldvalue` that can be passed freely between threads, easily serialised to disk, and cast safely to and from native C types. At the time of writing, I have implemented support for all of the following types:

- Integers (range of `stdlibc`'s `intmax_t`).
- Floating Point / Real Numbers (range of C's `long double`).
- String (null-terminated).
- Cloud Pointers (see section 3.6).
- Cloud References (references to any arbitrary CIEL ID).
- Arrays (for returning multiple `cldvalues`; item count can be queried).

Note that the support for arrays allows for nested `cldvalues`, e.g. arrays of arrays.

A cloud thread must return a `cldvalue` object and when the task exits, this `cldvalue` is serialised using cJSON into a `SWDataValue` reference that is committed to the local block store under the task's output ID. The task's output ID is then published to the master node, notifying it that the output has been evaluated and at the same time informing it of the location where it resides.

These `cldvalues` are therefore retrievable not only by blocking Cirrus tasks but also by SkyWriting tasks making them extremely flexible, but also safe from a type-checking point of view. In the former case, thread values can be queried by combining `cldthread_result()` (which returns a `cldvalue`) with an appropriate cast command, e.g. `cldvalue_to_long()`, `cldvalue_to_string()`.

3.5 Sample Program

By implementing `cldvalues`, Cirrus is able to successfully spawn, execute and collect values from Cirrus threads.

The code presented in Listing 3.2 demonstrates the core functionality of `libCloudThreads`, fully implemented. This extract was adapted from the working example `helloworld.c` (see section 4.2.2).

```
/* Initialise libCloudThreads, terminating upon error */
if( !cldthread_init() ) exit( EXIT_FAILURE );

/* Spawn 4 new Cloud Threads */
for( i = 0; i < 4; i++ )
  threads[i] = cldthread_create( fptr_hello_world, (void *)i );

/* Demand scheduling and block for them to complete */
cldthread_joins( threads, 4 );

/* Print out the output of each thread to stdout */
for( i = 0; i < 4; i++ ){
  cldvalue *result = cldthread_result_as_cldvalue(threads[i]);
  write( stdout, cldvalue_to_string( result ) );
}
```

Listing 3.2: Code extract showing how to spawn Cloud Threads using `libCloudThreads`.

3.6 Distributed Shared Memory

Implementing shared memory for requirement 5 (section 2.2.1) was one of the harder aspects of this project. My proposal suggested offering a command such as `shrdmalloc()` that would allow the user to allocate memory that could be later shared with the rest of the cloud.

To realise this, I introduced the notion of a *cloud pointer*. Cloud pointers are what Cirrus uses to allocate and access memory that should later be readable once a thread has terminated. Each cloud thread that creates a cloud pointer has a managed memory heap from

which cloud pointers are allocated from. When a cloud pointer memory block is allocated, it returns a `cldptr` object that encapsulates an offset into this shared heap, and a unique identifier that represents that specific heap. This `cldptr` object can then be used to uniquely identify and request data from the shared heap of other cloud threads.

The first thing that had to be implemented were the managed heaps. Introducing complex memory allocation routines was deemed superfluous to the intentions of this project, so a simple dynamically sizing memory block is allocated for each running task. Each heap has its own unique ID (a bijection between executed tasks and heaps) and a block of memory that it allocates space from. This heap's space is divided up into a collection of consecutively linked memory blocks, each of which has a header indicating whether the memory block is in use; and the size of the adjacent space (and therefore the offset to the next block).

When a `cldptr_malloc()` call is made, `libCloudThreads` jumps through the block headers in the heap and if a large enough free space is found, it is reserved with appropriate block headers, and a usable memory pointer is returned as a `cldptr` struct. `cldptr` objects can then be encapsulated as a `cldvalue` to pass it onto other threads. Otherwise, if on reaching the end of the private heap no suitable memory location was found, the heap is resized and a headered memory block is appended to the end of the chain.

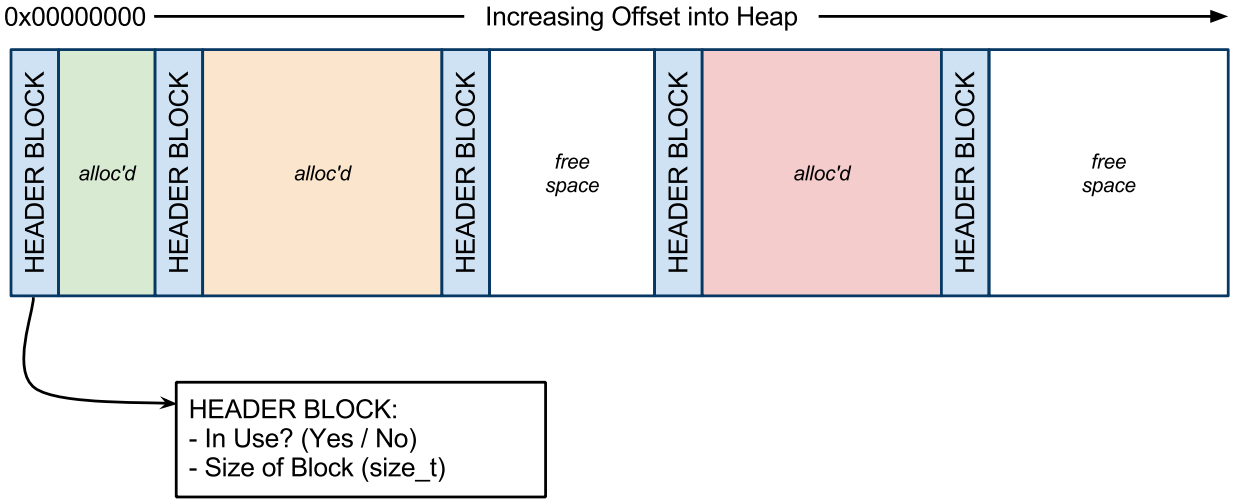


Figure 3.5: Layout of a cloud task's private `cldptr` heap.

Although `cldptr_malloc()` runs in $O(n)$ time (where n is the highest number of memory blocks allocated), the accompanying `cldptr_free()` command runs in $O(1)$ time, simply marking the flag in the memory block header as "free". This arrangement also means that the allocated heap space is monotonically increasing but a `cldptr_compact()` method could be easily implemented if desired.

If a task allocated any `cldptr` memory whilst it was running, when finished it saves its heap by dumping the contents to the local block store under a deterministically generated ID.

This CIEL ID is published to the master node to notify it of its location. The master will pass this location information on to any of the workers that are allocated tasks dependent on this ID.

The `cldptr` object contains a key that represents the heap ID it belongs to, along with the offset into that desired heap. This is important if a `cldptr` is attempted to be dereferenced in a task that it was not allocated by. When the user calls `cldptr_deref()`, the `cldptr`'s heap ID is compared with the current heap ID and if found to be the same (i.e. it was allocated by the same task) the local memory pointer is returned in $O(1)$ time. Otherwise extra levels of indirection are employed to determine a pointer in a "foreign" heap.

First, the heap ID is looked up in a hash table of all the cloud heaps that have been previously dereferenced by this task. If an entry exists, the hash-table value contains the base pointer of the cloud heap as it is mapped into the local task's memory. This base pointer is added to the `cldptr`'s offset to produce the memory location to return. By using a hash table, I ensured that memory access was still scalable with $O(1)$ time.

The other possibility is that the heap ID is not found in the hash-table and has not been accessed by the task previously. To load the heap, `cldptr_deref()` will first convert the heap ID into a CIEL ID. It then attempts to open this ID from the current worker's local block-store. If a copy of that heap is not available in the block-store, the task blocks – spawning a continuation task which is dependent on the heap's CIEL ID (see section 3.3). When the continuation task resumes, the heap ID is guaranteed to be available in the block store.

Once the heap has been located in the block store, it is mapped into memory with read-only permissions (using `stdlibc`'s `mmap()` function), and the base pointer is added to the hash-table. The execution time of this scenario is dependent on a lot of external factors such as task scheduling and network transfer rates but a rough estimate would suggest that it is $O(m)$ (where m is the size of the memory block). The use of `mmap()` makes the reading of large memory blocks into RAM relatively fast, as the OS often optimises the load by paging into memory only those sections being accessed.

A linked-list test case (later described in section 4.2.2) was implemented using `cldptr_malloc()` where several threads are used to construct a shared list that is later iterated over.

3.7 Proposed Extensions

This section details all of the extensions that I implemented from my original Project Proposal. This includes *future* synchronisation primitives and the streaming of I/O between threads.

3.7.1 Synchronisation Primitives

Instead of implementing distributed mutexes and condition variables which would have a large network synchronisation overhead, I intentionally designed the cloud values concept (in section 3.4) such that the user is able to create references to arbitrary CIEL IDs. This cloud value then represents a "lazy future" – a flexible primitive that maintains the framework's determinism [1].

These lazy futures can be waited on just like thread output CIEL IDs and can therefore be used to block on work that another thread might schedule. They can therefore be used to synchronise threads and as each Cirrus thread's memory space is isolated and private[1], there is no data contention issue.

3.7.2 Forwarding and Collection of I/O

Forwarding and collecting another thread's I/O provides a way for developers to distribute code that might traditionally use FIFO pipes in an intra-process environment.

In order to realise this, I was required to implement support for CIEL's streaming references (see section 3.8.1). This meant that I could open a thread's output as an append-only file-descriptor (revealed to the user by a call to `cldthread_stream_result()`). The code in Listing 3.3 will then remap the current thread's `stdout` to `cldthread`'s file descriptor.

```
int fd = cldthread_stream_result();
dup2( fd, STDOUT_FILENO );
```

Listing 3.3: Streaming `stdout` as the thread task's output

The forwarded I/O can be then collected using the `cldthread_result_as_fd()` command. This feature is demonstrated in the `prodcons.c` test where one thread task prints the output streamed from `stdout` of another task.

3.8 Additional Extensions

This section describes the extensions that I implemented in addition to those originally suggested in my project proposal.

3.8.1 Streaming References

A streaming reference is a CIEL concept that allows a task to stream data to another task without it having to wait for all the data to have been committed to the block-store first.

[1]This is because each Cirrus thread is executed in its own `blcr_fork()`'d process.

In order to do this, the worker's executor sets up named pipes that represent the data and continuously streams the file over HTTP from the worker that is running the streaming task.

As `libCloudThreads` publishes and commits data to the block store itself, I had to extend the library so that it followed the conventions expected by CIEL. To maintain simplicity, I decided that a thread can *either* stream its output *or* return a "concrete" cloud value once it has finished.

If a thread task calls `cldthread_stream_result()`, `libCloudThreads` notifies the CIEL master node that the task will be streaming its output. This permits any thread task that is blocking for this thread's output (i.e. requested a join) to be freely assigned to a worker. The Cirrus executor was extended so that, once assigned, it passes the path of these named pipe streams to `libCloudThreads`. This named pipe is opened immediately which allows CIEL to stream output from the streaming task's worker. After returning from `cldthread_joins()`, a file descriptor to this named pipe is requested using `cldthread_result_as_fd()`.

Streaming references allow Cirrus apps to be able to work with large data sets, but also allow the parallelisation of data sets that have no explicit size. I exploited this feature when implementing the I/O forwarding in section 3.7.2.

3.8.2 Memoisation

One of the test cases that were implemented to verify that cloud threads could spawn cloud threads themselves was a Fibonacci Number Generator (later described in section 4.2.2). While it would have been of course trivial to implement the Fibonacci Number Generator using an $O(n)$ algorithm, this recursive implementation was written as a stress test for hierarchical thread spawning.

As can be seen in figure 3.6, this implementation spawns hundreds of tasks by evaluating $Fib(6)$ or $Fib(7)$. To calculate the upper-bound on the number of tasks this implementation would spawn, I will define a function $T(n)$ that counts the number of recursive calls required to calculate $Fib(n)$.

$$T(n) = \begin{cases} T(n-1) + T(n-2) + 1 & n \geq 2 \\ 1 & n = 1, 0 \end{cases}$$

By recursively expanding the $T(n-2)$, $T(n-3)$, ..., terms and noticing that the resulting coefficients are themselves Fibonacci numbers, we obtain:

$$\begin{aligned}
T(n) &= & T(n-1) &+ & T(n-2) &+ & 1 \\
&= & 2 \cdot T(n-2) &+ & T(n-3) &+ & 1+1 \\
&= & 3 \cdot T(n-3) &+ & 2 \cdot T(n-4) &+ & 1+1+2 \\
&= & 5 \cdot T(n-4) &+ & 3 \cdot T(n-5) &+ & 1+1+2+3 \\
&= & 8 \cdot T(n-5) &+ & 5 \cdot T(n-6) &+ & 1+1+2+3+5 \\
& & \vdots & & \vdots & & \vdots \\
&= & Fib(n) \cdot T(1) &+ & Fib(n-1) \cdot T(0) &+ & \textstyle\sum_{i=1}^{n-1} Fib(i)
\end{aligned}$$

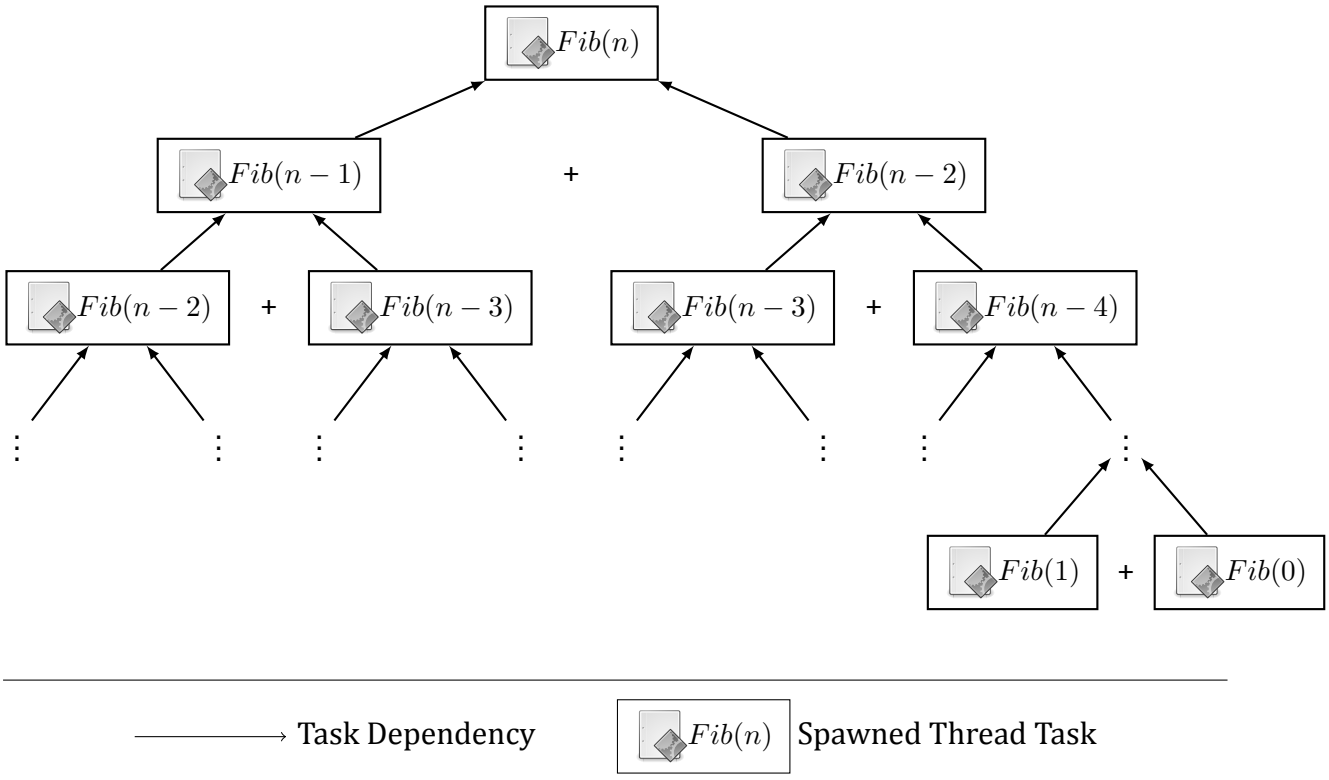


Figure 3.6: Cloud thread task dependency graph for recursive $Fib(n)$.

Substituting the values for $T(0)$ and $T(1)$ gives the following simplified expression:

$$T(n) = Fib(n) + Fib(n-1) + \sum_{i=1}^{n-1} Fib(i)$$

$\sum_{i=1}^{n-1} Fib(i)$ can be shown to be equal to $Fib(n+1) - 1$ using proof-by-induction[2].

$$\begin{aligned} T(n) &= Fib(n) + Fib(n-1) + [Fib(n+1) - 1] \\ &= Fib(n+1) + Fib(n+1) - 1 \\ &= 2Fib(n+1) - 1 \end{aligned}$$

To obtain an upper bound, first rewrite $Fib(n)$ in terms of the golden ratio[3], φ:

$$2Fib(n+1) - 1 = 2 \cdot \left\lfloor \frac{\varphi^{n+1}}{\sqrt{5}} + \frac{1}{2} \right\rfloor - 1 \tag{3.1}$$

Expressing equation 3.1 using big-O notation gives:

$$\begin{aligned} O\left(2 \cdot \left\lfloor \frac{\varphi^{n+1}}{\sqrt{5}} + \frac{1}{2} \right\rfloor - 1\right) &= O(\varphi^{n+1}) \\ &= O(\varphi^n) \end{aligned}$$

Therefore, with the original implementation Cirrus could execute recursive $Fib(n)$ in $O(\varphi^n)$ time, spawning and blocking on $O(\varphi^n)$ threads. But, of course, this is the result of the recursive $Fib(m)$ being redundantly evaluated several times (where $0 \leq m \leq n$). This evaluation is inefficient since as soon as the first $Fib(m)$ task completes, its result is available to any worker on the cloud, which prompted investigation into how this redundancy could be mitigated.

By making the output ID of Cirrus threads deterministic on their input parameter instead of being arbitrarily unique, threads can be offered immediately the value of previously scheduled Cirrus thread computations. This acts as a form of memoisation, and can offer significant speed-ups when executing independent tasks that share common generated data.

In order to achieve this, I introduced the notion of a *smart thread* where, in addition to the `fptr` and `arg` that a standard cloud thread requires, the user can also specify a *calculation group* ID. By passing a non-`null` calculation group ID to the `cldthread_create_smart()` function, the cloud thread task would be spawned with deterministic task and output IDs – generated solely using the thread's `fptr`, `arg` and calculation group ID[4].

When the memoisation task is submitted by `libCloudThreads`, the CIEL master drops any tasks with identical IDs that were currently waiting and as such only one uniquely prescribed invocation from a calculation group will ever be executed. The spawning thread

[2] See appendix A.1.

[3] Using the computational equivalent of Binet's formula which makes use of the floor function.

[4] The purpose of the *calculation group* is to minimise function pointer clashes as C compilers may often layout different program code in similar ways.

is still returned the deterministic task thread ID and can `cldthread_join()` it just like any other thread.

The recursive Fibonacci Number Generator implementation perfectly demonstrates the effectiveness of this feature. The code was rewritten to use `cldthread_smart_create()` with an arbitrary constant set as the *calculation group*. This results in the number of tasks actually scheduled for execution being significantly reduced (see figure 3.7), slashing the running time from $O(\varphi^n)$ to just $O(n)$.[5] However, Cirrus memoisation could be used to improve the overall execution speed of any Cirrus program whose threads share repeatedly generated results. Incremental processing [7] performed using Cirrus would also benefit from this extension.

3.9 Redundant Extensions

The proposed extension of providing a fall-back to system scheduled threads was intentionally avoided. It is important to note that the `fork()` and `join()` technique (see section 3.2.1) employed by Cirrus results in cloud threads having isolated memory spaces which is completely different to the shared memory view that system threads have.

Although I did look into the potential of implementing a Unix `fork()` back-end to Cirrus, it quickly became apparent that doing so would limit the main feature-set of `libCloudThreads` with little benefit. For example, the ability of a task to wait for the output of a child thread that has not yet been created is possible using a CIEL back-end but would have to be coerced using complicated logic when running under Unix `fork()`.

By dedicating the library to the scheduling of *distributed* tasks, Cirrus was able to simplify a lot of aspects that might otherwise have to be exposed to the user. `libCloudThreads` is however able to notify a program when it is initialised if a cloud framework is not available, after which the user is then free to implement code using traditional threading routines himself. And of course, one is still able to execute Cirrus distributed tasks entirely locally simply by launching multiple CIEL worker processes (as shown later in section 4.2.1).

3.10 Miscellaneous Work

3.10.1 Profiling

In order to evaluate `libCloudThreads`, I had to implement a means for the code to profile itself without embedding profiling code too deeply into the source files. Most of the free tools already available (such as `gprof`) would only record the amount of time elapsed in CPU user mode which meant that lengthy I/O operations or system calls are not accounted for in

[5] This is later evaluated in section 4.4.3.

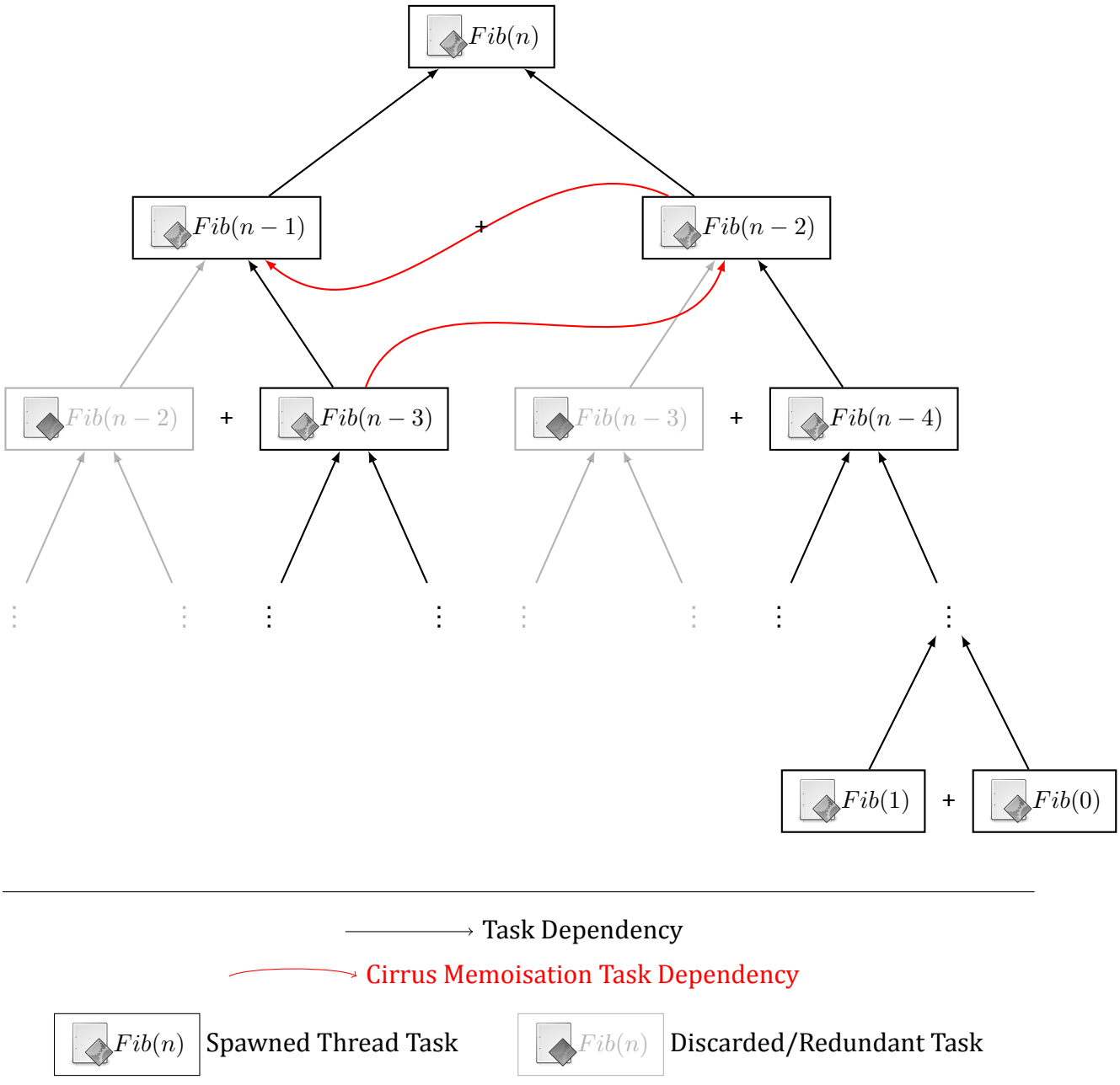


Figure 3.7: Cloud thread task dependency graph for a memoised recursive $Fib(n)$.

the results. As the longest operations in `libCloudThreads` involve the transmitting of data over a network, and checkpointing a process out to disk, these tools simply don't return indicative results. Other C profiling solutions either require modules to be compiled and injected into the kernel; would require the task executors to be modified so that they start programs through a boot-strap binary (such as `valgrind`); or are interactive (such as `gdb`). None of these solutions were plausible to use in a distributed environment with potentially hundreds of remote worker nodes.

As such, I wrote a C profiling module that, when linked with `libCloudThreads`, is able to record the execution time and call-stacks of each function and output it to disk. I designed it so that to benchmark any function, I simply add a `TIMER_LABEL( func_name())` preprocessor directive to the top of a function and include a `timer.h` header file.

This has no effect on the semantics of the program or the generated assembly code unless the `PROFILE` directive is defined. This occurs when `libCloudThreads`'s `makefile` is executed using the `profile` target – the library is then compiled with the `-finstruments-functions` GCC flag which inserts calls to stack tracing routines in the profiling module whenever the program enters or exits any function.

When a function marks itself using `TIMER_LABEL( func_name())`, its name is associated with the current timing session in the stack and when it exits, it then writes out the timing information of the complete function call to a CSV[6] file. The timing information in this file can then be retrieved and interpreted at a later convenient time.

3.10.2 Scripting

The development of this project required the writing and execution of scripts using several different languages. Namely, SkyWriting scripts had to be written for each of the test cases as, when the project was forked, this was the only available means for submitting a job to the CIEL framework.

In addition to this, while CIEL does provide a few scripts for managing CIEL clusters, I had to write a plethora of bespoke Bash scripts for installing `libCloudThreads` on nodes, and for scheduling and executing tests across several nodes on Amazon EC2, but also when managing several concurrent instances on my local development machine.

Finally `makefiles` had to be generated and maintained so that the project and its test suite could be compiled remotely using a script (without the development IDE).

3.11 Summary

In this chapter, I outlined how the core functionality in Cirrus was implemented and how the `libCloudThreads` library was built from the three major components. I explain how

[6] Comma Separated Values – an industry standard, application neutral, tabular file format.

the BLCR process checkpointing library was used in conjunction with the CIEL distributed execution engine to spawn threads on the cloud, and explained how I tackled the idea of passing C-based values in a distributed environment.

In section 3.5, I show how the core Cirrus API (documented in appendix B) can be used to spawn cloud threads that successfully run and execute on a CIEL network. By doing so, Cirrus realised all of the original project's requirements from section 2.2.1:

Requirement 1 (✓)

The framework must provide a way to execute a function on an arbitrary node of a distributed cluster. This should be done with the abstraction of a "cloud thread" which uses the function as its entry-point.

Cloud threads can be created and spawned from an entry-point-representing function-pointer using `cldthread_create()`.

Requirement 2 (✓)

The framework must allow the aforementioned entry-point to accept input from the caller (e.g. through at least one argument).

Cloud thread entry points accept one (`void *`) argument but can also access any other memory written to before the cloud thread was created.

Requirement 3 (✓)

The framework must be able to block and wait for a specific cloud thread's return value.

The `cldthread_joins()` command allows the current thread to wait for (and schedule if necessary) the evaluation of another thread on the network. Its return value can be obtained using `cldthread_result()`.

Requirement 4 (✓)

The framework must allow threads themselves to schedule and block on the output of other threads both arbitrarily and independently.

Cloud threads are first-class citizens in that they too have exactly the same ability to spawn threads and wait on them just like the original execution. This will later be demonstated in chapter 4.

Requirement 5 (✓)

The framework must provide a means for threads to share data between themselves.

Cloud threads can share data by returning cloud values (strings, integers, arrays, floating point numbers, etc.), streaming data to one other, or by making use of the distributed memory allocation routines provided by the cloud pointer abstraction (section 3.6).

I then went on to describe the extensions that I undertook, some of which were suggested in my project proposal, others of which were conceived during the implementation period. These extensions further improve Cirrus, making it more flexible than the original proposal dictated, and taking better advantage of CIEL's feature-set to offer significant speed-ups and performance gains under certain situations.

Finally, I briefly discussed the scripts and profiling framework that I was required to write to properly test and evaluate Cirrus, the results of which I will go on to present in chapter 4.

Chapter 4

Evaluation

The purpose of this chapter is to evaluate the success of Cirrus using test cases, present any results and outline how they were obtained as well as to suggest areas where Cirrus could potentially be further extended or improved.

4.1 Overview

Each of the criteria outlined in my original proposal (see appendix C) were completely satisfied, as detailed below.

Criterion 1 (✓)

A plan for distributing a process amongst nodes (inc. program code, state and memory).

This was investigated in chapter 2 and finalised in chapter 3. The result was a plan that was flexible enough to accommodate all of the proposed ideas and even a few advanced extensions that could provide significant speed enhancements under certain circumstances.

Criterion 2 (✓)

Implementation of the plan by extending the CIEL framework to accommodate, e.g. process-level checkpointing.

This part of the implementation was fully completed as described in section 3.2.1 and section 3.2. Processes are able to checkpoint themselves and CIEL task executors were written to run Cirrus tasks on the cloud.

Criterion 3 (✓)

An implementation of an API that is flexible enough to schedule cloud, or potentially system threads, transparently to the programmer.

An API was designed, documented (see appendix B) and exposed to the user as described throughout the latter sections of chapter 3. The library is able to report to an application if the Cirrus environment is not available, allowing the user to fallback to standard system thread routines if desired.

Criterion 4 (✓)

A set of parallel algorithms should be written and used to use in development; as test-cases; and in verifying that the solution is working.

Test programs were used as a vital development tool and whenever changes were made, regression testing was performed. The set of test-programs that were written are documented in section 4.2 and were each designed to test specific functionality or behaviour in `libCloudThreads`. At the time of writing, each test program is able to successfully execute and produces its desired result.

Criterion 5 (✓)

Tests should be performed to determine the speed and scalability of the resulting solution.

Cirrus was evaluated using the tests developed for *Criterion 4*. Test data was obtained using both services provided by the Amazon EC2 cloud network and my development virtual-machine.

Although some parts of the project inevitably took longer than expected, this was mainly absorbed by milestones either side that were completed more quickly than originally planned. For the most part, I was able to stick to the milestones set out in project proposal (see appendix C).

The only part that took significantly longer than expected was the evaluation of Cirrus, mainly due to the migration of testing to Amazon EC2. This was due to the difficulties of compiling BLCR with the custom hypervisor kernels used by EC2 instances, but also due to the downtime the service experienced in late April. Thankfully, both of these issues were soon resolved without delaying the rest of the project too severely.

4.2 Testing

Testing was used extensively throughout the development process, with regression testing being carried out whenever a major milestone was reached. Test cases were constructed as and when they were required and written entirely in C using the API provided by `libCloudThreads`.

4.2.1 Testing Environments

Two environments were use to test and evaluate Cirrus. Initially, a virtual cloud was set-up using the local-loopback facility on my development machine. But once the implementation stage had concluded, `libCloudThreads` was evaluated on Amazon EC2 – a fully operational commercial cloud environment. The schematics of both CIEL environments are shown in figures 4.1 and 4.2.

The local cloud environment runs a master and two worker processes concurrently. Each process is assigned its own separate block store on the same local disk, and listens for incoming HTTP requests on unique ports. A job console is used for submitting jobs and waiting to display the output. The OS routes all network packets between these processes internally, i.e. they never leave the computer, which means communication is fast. No more than two worker processes are typically run as the development machine only offers a dual-core processor.

Figure 4.1: Local CIEL *cluster, as set-up on the development machine.*

This is in contrast with the Amazon EC2 cloud environment. Each HTTP request is sent between nodes using IP routing. Each instance has its own block store but the entire root file system is mounted on a remote Amazon EBS volume where throughput is throttled by network speed and latency. However, each cloud node has only one worker process running so it can dedicate all resources to the one assigned task. The major advantage to the Amazon EC2 environment is that more instances are available on demand and so it can be used to see how well Cirrus scales.

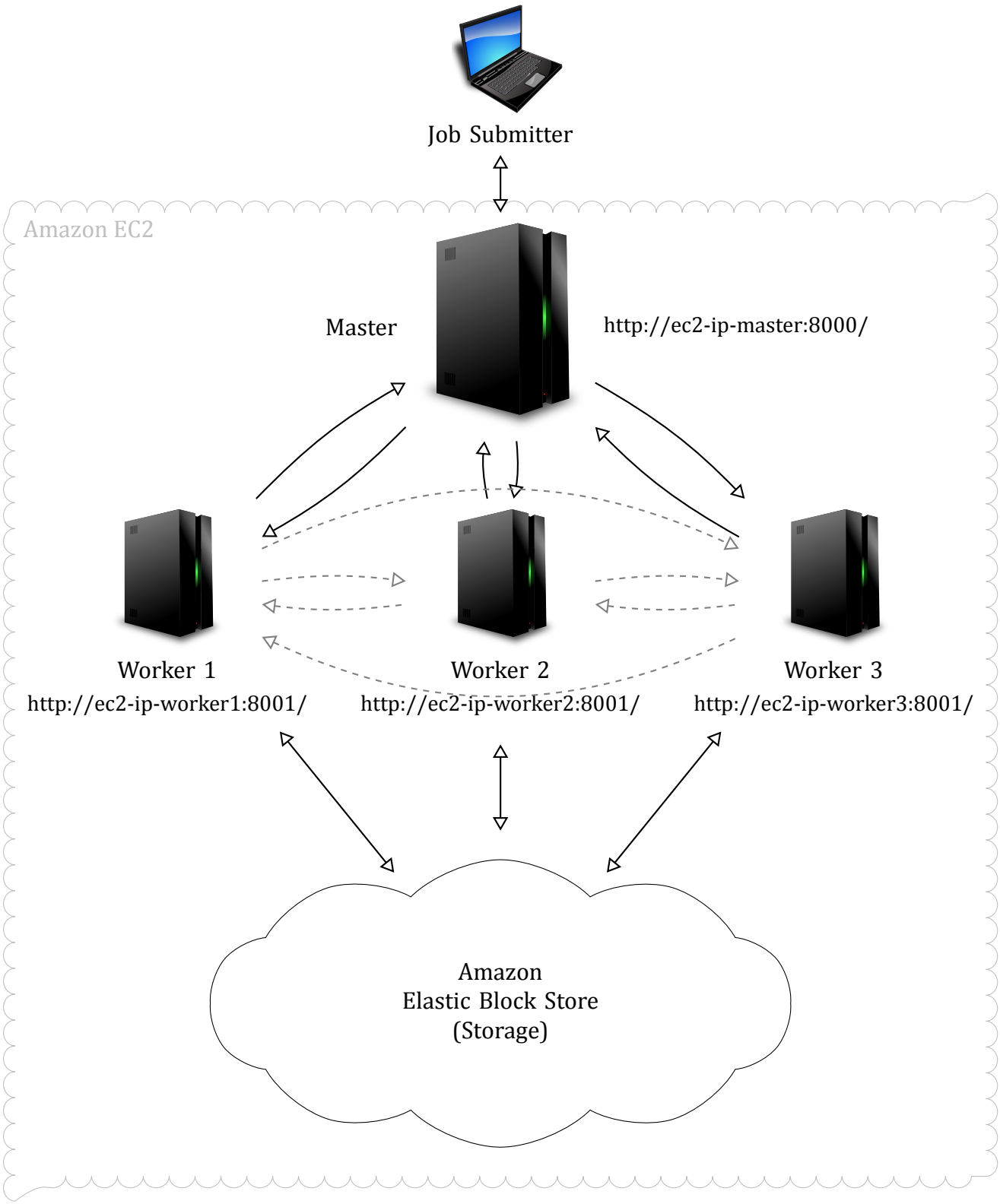


Figure 4.2: Distributed CIEL *cluster on the Amazon EC2 network.*

The hardware specifications of the respective systems are given in section A.2.

4.2.2 Test Programs

This section presents a list of the programs written to develop, test and evaluate `libCloudThreads`. The programs in Cirrus's test suite were designed to represent a range of different scenarios. For the purposes of this evaluation, each result obtained is the average result once the test case has been run four times. Whenever a test is run more than once on the same cluster, all CIEL nodes have their block stores flushed to stop any previously calculated results from being offered up without having to do the work again.

Hello World `helloworld.c`

This was the first test program that `libCloudThreads` was used to run and was designed to test the basic API of spawning threads, passing in and returning values, and blocking on their output.

It spawns four threads using `cldthread_create()`, passing each one an identifier through an argument. The app then `cldthread_joins()` the spawned threads, blocking the main task and forcing evaluation of the child threads. When executed, each child thread prints out "Hello World" ten times with a short delay, simulating a lengthy calculation, and then returns a self-identifying string. Once all the child threads have executed, the parent thread collects all the output strings together, concatenating the strings and returning them as the program output.

This was later adapted when `cldvalue` (section 3.4) introduced support for arrays, consolidating the thread output as an array of values instead of a single concatenated string.

Pi Calculator `pi.c`

This test program was the first Cirrus application that was able to perform what one might consider meaningful work (using a Map / Reduce paradigm). The program uses a Monte Carlo simulation to estimate the value of π.

The total number of iterations is specified as an argument on the command line, along with the number of desired worker threads to split the work between. In order to isolate a particular deterministic set of random number for each thread, I implemented a *Halton Sequence* (inspired by a similar test-case used for CIEL benchmarking). A final cloud thread performs a "reduce" operation aggregating each thread's result and outputting a value for π.

Fibonnacci Number Generator `fib.c`

As perhaps the most notable test case, this recursive program was originally written to stress `libCloudThreads` and to ensure that child tasks could spawn and wait on tasks themselves. Whilst it achieved its original purpose, its running time and inherent inefficiency inspired the implementation of the memoisation extension described in section 3.8.2. This

test case was the first adapted to leverage the *smart thread* memoisation API, reducing its running time from $O(\varphi^n)$ to just $O(n)$.

Producer / Consumer [Streaming Output] `prodcons.c`

This test was written during the implementation of streaming outputs in `libCloudThreads`. A producer task is spawned which maps `stdout` to the thread's streaming output. The producer proceeds to write messages which are read asynchronously by a remote consumer that prints the output. In addition to demonstrating the redirected I/O extension (section 3.7.2), this also stress-tests Cirrus's interaction with the protocol employed by the CIEL block stores, and ensures that `libCloudThreads` issues task dependencies correctly such that two tasks can be scheduled in parallel.

Linked List `linkedlist.c`

This test was written to demonstrate the shared memory aspect of `libCloudThreads` which was implemented in section 3.6. A shared integer linked-list is created by spawning several tasks, each of which append an arbitrary value to the head of a list. The complete list is then returned by pointer to the main task. It then iterates over all the links – retrieving; mapping into memory; and decoding the serialised heaps from all of the workers on the fly – printing the value of each item as proof.

Mandelbrot Set Generator `mandelbrot.c`

This final Cirrus application was written to test the handling of binary data by the `libCloudThreads` framework. Dimensions, scales and offsets into the Mandelbrot set are passed as input parameters and the application then divides the region into a set of tiles. A worker is given a task to generate each tile, which is then output as a PNG file. The individual tiles from the worker node are then stitched into the complete image, which is returned as the output.

4.3 Project In Use

The aim of Cirrus was to create a framework that would allow legacy applications to harness the computing power available to them on a distributed computing platform.

Figure 4.3 presents the running times of the "Pi Calculator" test program running on an Amazon EC2 cluster. The graph demonstrates `libCloudThreads`'s ability to harness the computing power of differently sized clouds. Increasing the number of threads on a one-worker cluster resulted in a linear increase of 0.45s per thread which represents the overhead of spawning and joining Cloud Thread tasks. However, by increasing the size of the cluster to two, three, five or even ten nodes, execution time was cut by up to 90%.

Local minima occur periodically when the number of cloud threads is an integer multiple of the number of workers. As expected, each cluster experiences the shortest running time

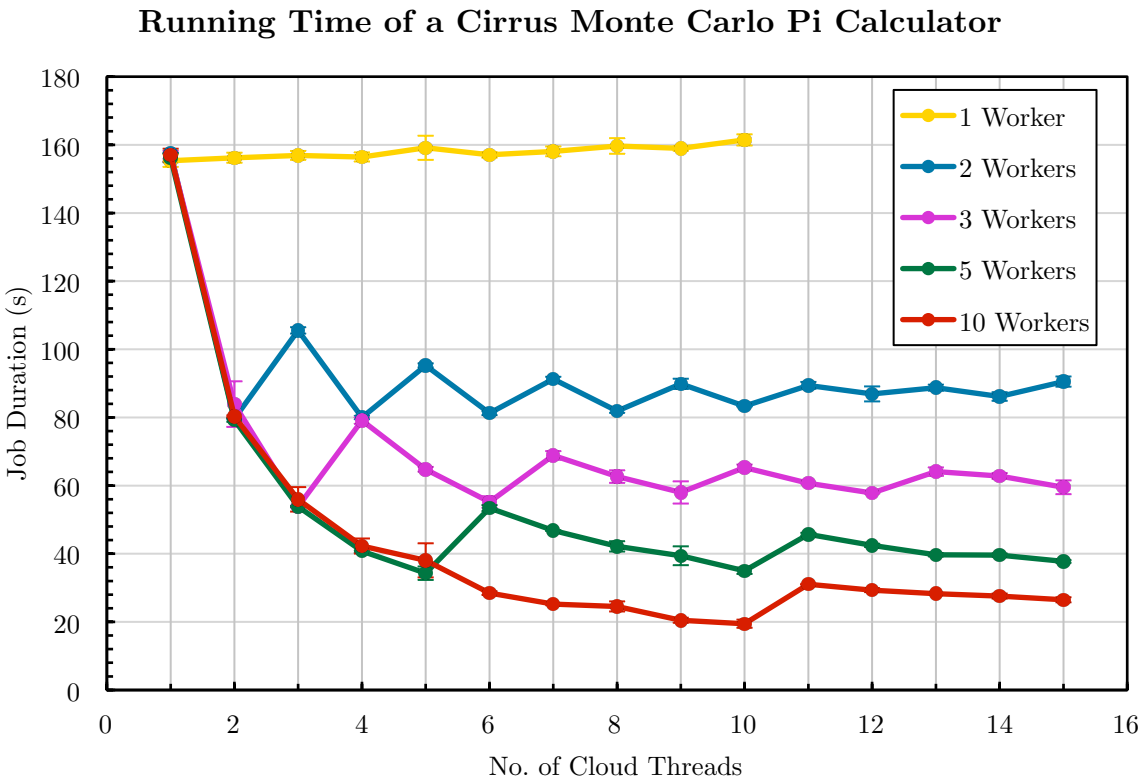


Figure 4.3: The time taken to calculate π by equally splitting 3×10^9 iterations of a Monte Carlo simulation between a varying number of Cirrus cloud threads. Error bars show $\pm\sigma$. Each value is averaged from 4 runs.

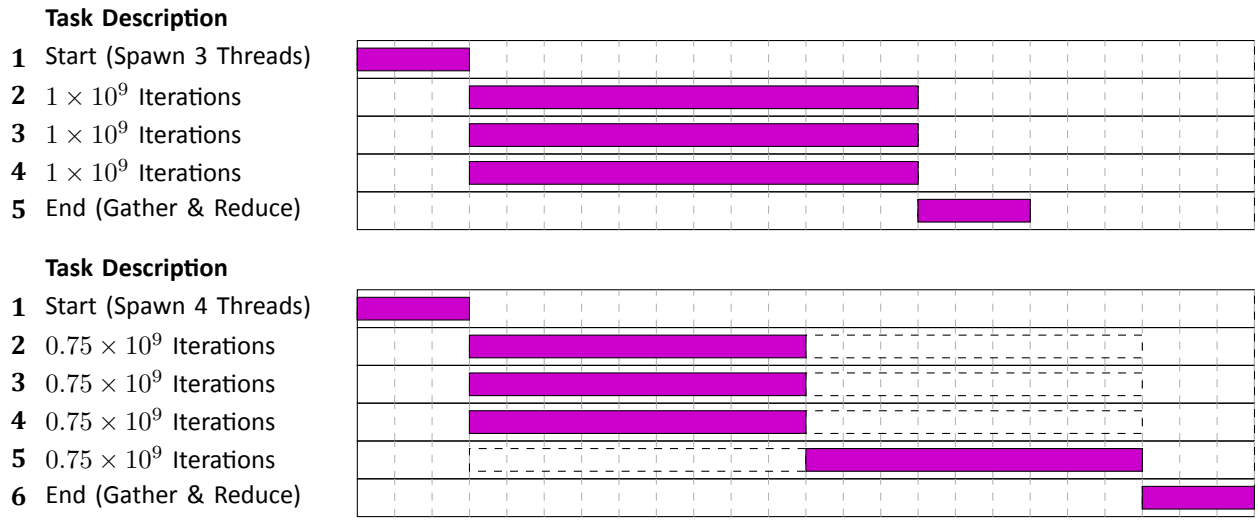


Figure 4.4: Gantt Chart representing `picalc.c`'s cloud thread tasks when spawning three (above) and four (below) threads on a three-worker cluster.

when the number of available worker nodes is equal to the number of cloud threads spawned, where no threads have to wait.

The immediate increase in running time that subsequently follows a minima is because only one worker can work on a task at a time. With fewer workers than tasks, some will have to complete an additional task whilst others remain idle (see figure 4.4).

This phenomenon is apparent because all tasks in this example take an *equally long* period of time to complete. As the number of spawned threads increases, the time taken to complete each task falls. This increase in granularity allows any surplus work to be shared out more evenly amongst idle nodes. This is why the spiking in figure 4.3 becomes less pronounced as the number of threads increase.

Fixed thread-length programs represent only a small subset of the applications that `libCloudThreads` can be put to. A more generalised example is demonstrated by `mandelbrot.c`. Figure 4.5 shows the speed-up Cirrus can offer when distributing a variable parallelisable algorithm.

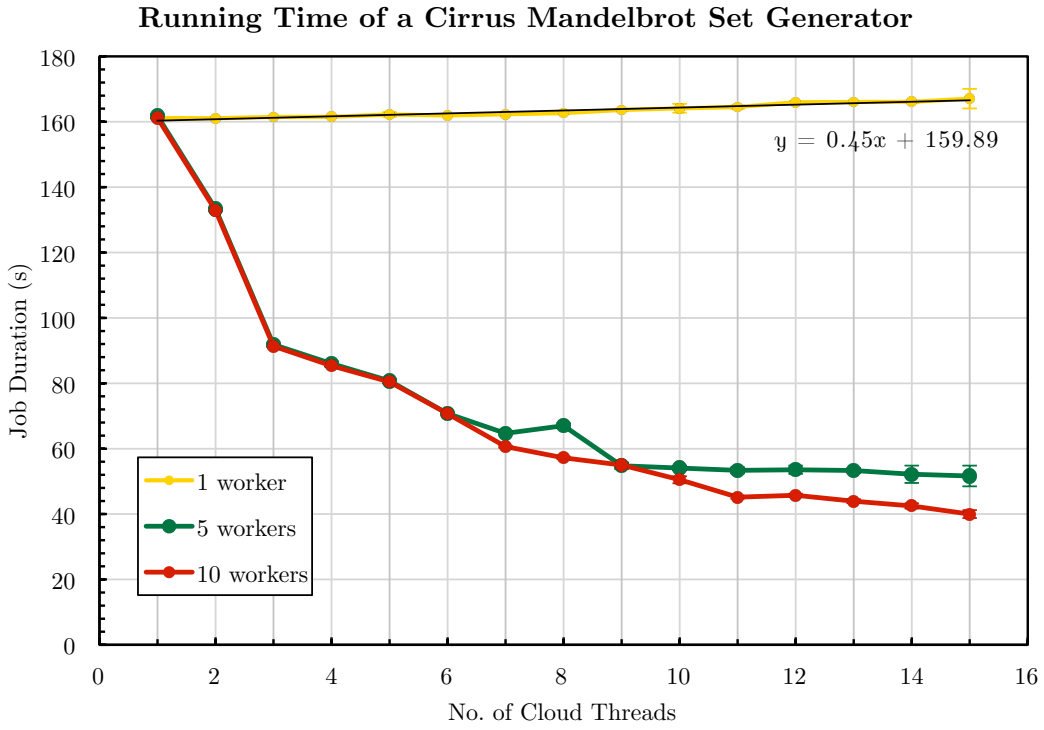


Figure 4.5: The time required to generate a 10^8 pixel image using `libCloudThreads` on a variety of differently-sized cloud clusters. The generated image is depicted in figure 4.6. Error bars show $\pm\sigma$. Each value is averaged from 4 runs.

The job was parallelised by splitting the image (figure 4.5) into equally spaced strips from the left edge to the right. Unlike `picalc.c`, this simulated a large variety of task lengths as each strip requires differing totals of Mandelbrot iterations ($\propto$% of strip that is red).

Without parallelising the algorithm at all, the average running time was ≈160s. However, by increasing the CIEL cluster size to five or even ten workers, the test experiences significant speed gains from using Cirrus.

The varied task lengths result in the execution times converging to a steady medium far sooner than they do in the rigid `picalc.c` example. However, the running time plateaus off at around 50s in both the five and ten worker set-up as the duration of the final tile-stitching task becomes the limiting factor. This results in a minimal difference in execution time when running on a five and ten worker CIEL cluster for this implementation, but a more optimised stitching algorithm could be employed to allow this example to scale further.

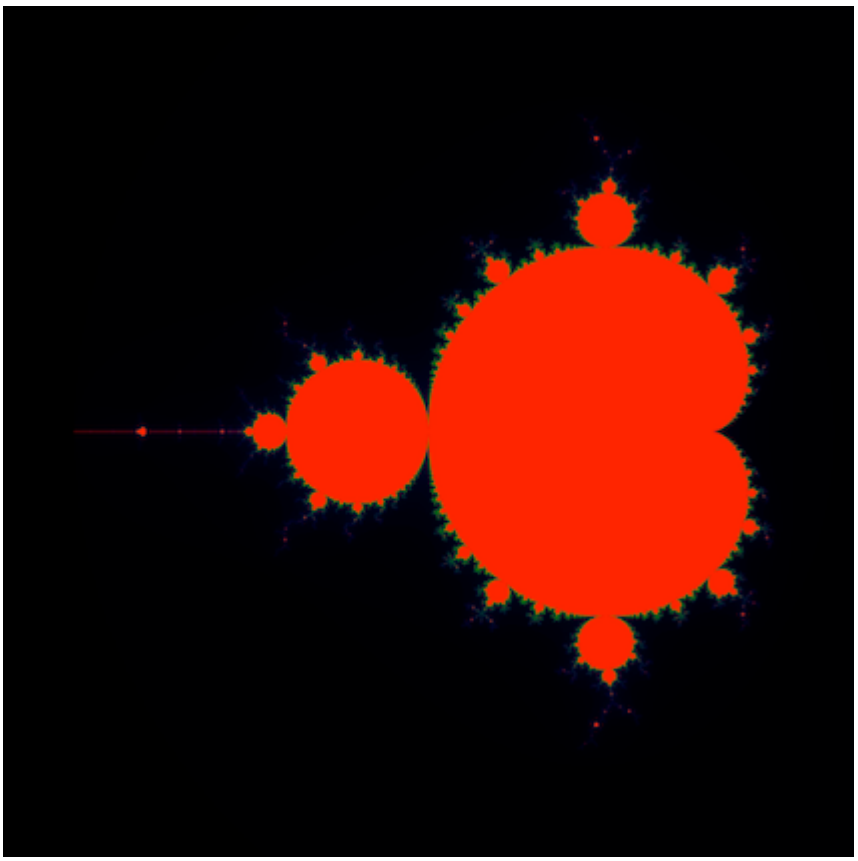


Figure 4.6: A scaled down version of the image produced by the test performed in figure 4.5. The image represents the Mandelbrot set between $(-0.75, 0) \pm (1.375, 1.375)$ *on a* $(\mathbb{R}, \mathbb{I})$ *plane, with a maximum of 1,000 iterations (represented using bright red colours).*

4.4 Performance & Scalability

4.4.1 Spawning a Cloud Thread

To measure the performance of thread spawning, I recompiled `libCloudThreads` using the profile module that I had implemented in section 3.10.1. This was important for getting meaningful results as `libCloudThreads` performs a lot of I/O operation, both writing to disk and sending/receiving over the network.

Figure 4.7 compares the average time taken to spawn a new cloud thread on both an Amazon EC2 instance and my development virtual machine.

The first thing that should be noted is that the total amount of time required to start a task on an Amazon EC2 instance is almost twice that when spawned on my development machine. This is most likely because of the bottleneck that the EC2 instance is subject to when reading/writing to its network mounted file system. BLCR writes its checkpoint output to disk, but also `libCloudThreads` writes out JSON encoded serialised CIEL references that are required for the submitted task descriptor. Also, network traffic is expected to be slower in a real cloud environment where packets are sent over communication channels rather than retained internally by the OS local-loopback.

The other important discovery that I found was that in both cases the process checkpointing operation performed by BLCR is responsible for between 97-98% of the total thread spawning time.

The checkpointing process takes up a significant portion of the overhead in spawning a Cloud Thread. The time that BLCR takes to complete a checkpoint is dependent upon many

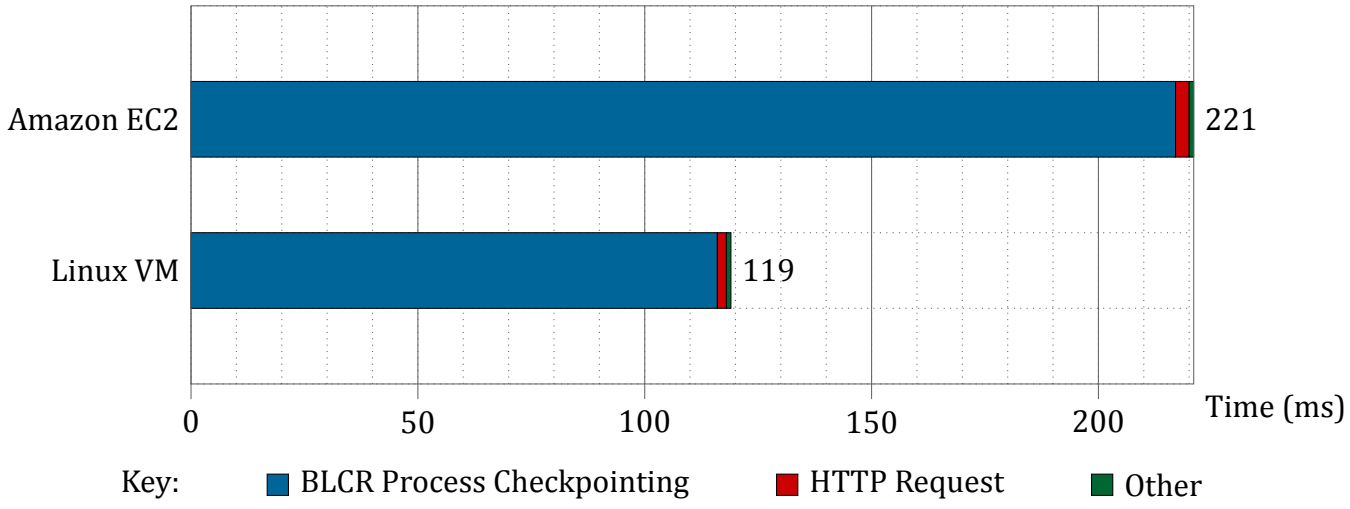


Figure 4.7: Time taken to create a Cirrus cloud thread in the `picalc.c` test case. Each value is averaged from 4 runs.

factors, such as the size and memory footprint of the executable, not to mention the system's disk access speed.

BLCR does however offer a few options that determines how deep a copy it makes of a running process. `libCloudThreads` was optimised for the common case by only checkpointing the program code, main memory and any `mmap()` files (such as dynamically loaded shared libraries or `libCloudThreads`'s cloud heaps in section 3.6). This assumes that any other files open are available at the same path on all cloud nodes; however BLCR can be set to include this data at the expense of an even larger overhead.

However, the checkpointing operating still takes a considerable amount of time so in section 5.3, I list this as one of the areas that could benefit from further work.

4.4.2 Distributed Shared Memory

My intention was to have the access times for reading shared heap data scale well with the number of shared heaps that a task (or job) required. Figure 4.8 shows the time taken for `linkedlist.c` to spawn, create, serialise, transfer, and mount into memory n shared heaps of similar size (one link of an integer-valued list).

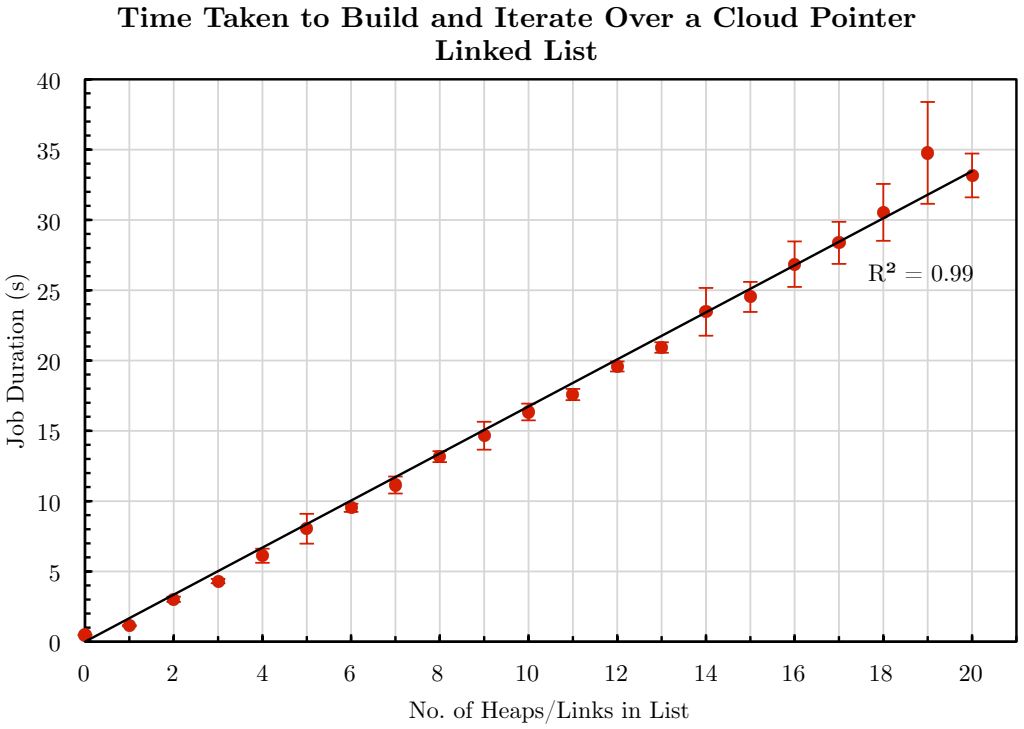


Figure 4.8: The time taken to distribute and access n different Cloud Pointer heaps on a 10-worker CIEL *cluster. Error bars show $\pm\sigma$. Each value is averaged from 4 runs.*

The data corroborates my intentions with running times scaling linearly with the number of heaps accessed. Meeting this target means that programmers should feel comfortable using Cirrus's shared heaps in their C programs without compromising too much on speed.

4.4.3 Memoisation

The theoretical speed-ups provided by Cirrus's memoisation extension can be quite profound depending on the algorithm that is exploiting it. Figure 4.9 shows the execution time of the intentionally inefficiently recursive `fib.c` being cut from exponential to linear time on a one-worker Amazon EC2 CIEL cluster.

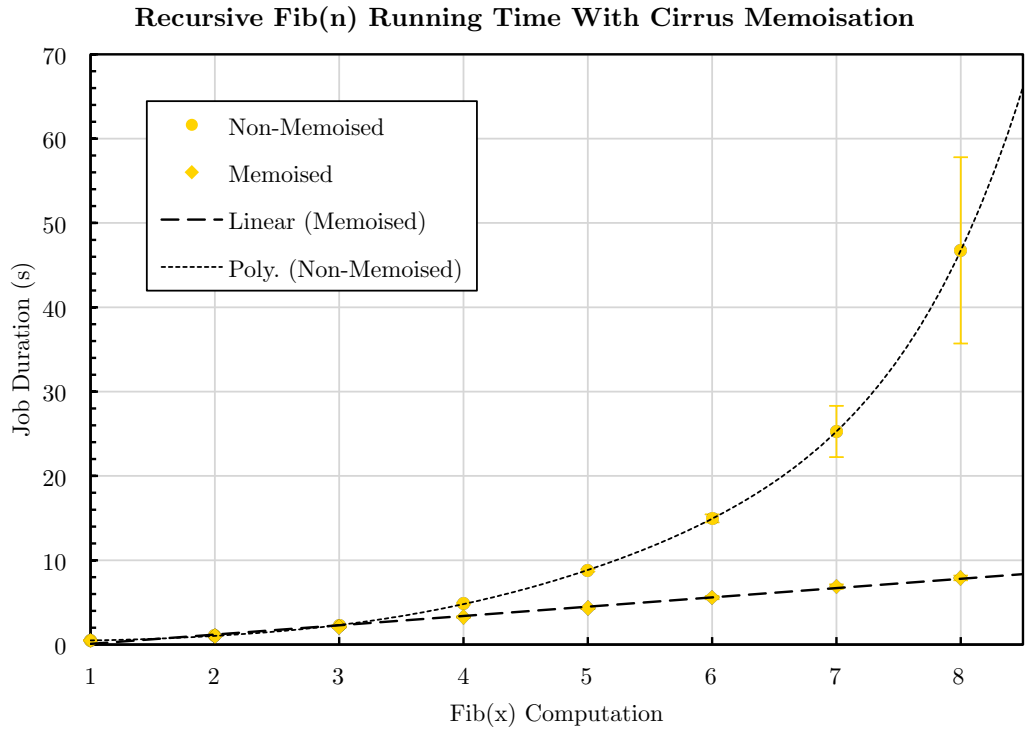


Figure 4.9: Comparison of the running times for a recursive $Fib(n)$ algorithm when executed on a one-worker cluster with and without Clouth Thread memoisation. Error bars show $\pm\sigma$. Each value is averaged from 4 runs.

This data confirms the theory presented in section 3.8.2. Memoisation is shown to be an effective way of dramatically reducing the running time of an algorithm in cases where scheduled threads share calls to common deterministic functions. The memoisation extension appears to successfully allow developers to effortlessly leverage the functionality of a distributed computing platform.

4.5 Limitations

Cirrus was designed to allow programmers to easily distribute *lengthy* operations in an attempt to reduce the overall running time. As shown earlier in figure 4.7, the time taken for Cirrus to checkpoint a process and spawn a new task can be ≈ 250ms. This overhead means that Cirrus is not a viable solution for attempting to speed up operations which take less than this amount of time. As with almost any computational framework, in order to reap the benefits of distribution, cloud tasks should be computationally intensive ($\gg 250$ms in this case).

In addition, developers should not expect Cirrus to magically distribute previously-written multi-threaded applications with no modifications. The aim of Cirrus was to provide a fast, light-weight framework that would provide effective scheduling and integration of *cloud threads* on a distributed network. While the API was likened to common threading libraries for the purposes of familiarity, cloud threads are not supposed to behave in exactly the same way as traditional system threads. For example, unlike the DSM projects discussed in section 1.2, Cirrus avoids heavy-weight emulation layers for sharing process memory space. Instead, cloud threads share data using return values, streaming outputs, or via the distributed memory heaps implemented in section 3.6.

4.6 Summary

In this chapter, I first explained how each of the proposed success criteria for Cirrus were met. I then went on to describe the test cases constructed and subsequently used them to quantitatively evaluate the scalability of the framework. I presented data which show Cirrus successfully distributing work as cloud threads on the Amazon EC2 commercial cloud service and how, in doing so, reduced the execution time of a selection of different algorithms.

I then use micro-benchmarks to look at the performance of Cirrus and present evidence that the *distributed shared heap* scales as I had hoped in section 3.6. The memoisation extension I implemented was also confirmed as introducing considerable savings on the running time of the recursive Fibonacci implementation and it is hoped this will offer speed-ups on the execution of other types of algorithm too.

Finally, I briefly discussed what should not be expected of Cirrus.

Chapter 5

Conclusion

This chapter concludes the dissertation having described the complete preparation, design, implementation and evaluation of the Cirrus framework.

Despite the challenges that came with working with a brand new research project and tackling and accommodating the nuances encountered when writing C, I must admit that I found the entire project extremely satisfying from both an implementation and research perspective. I thoroughly enjoyed working on an active research project with supervisors who showed such enthusiasm and were able to provide deep and valuable insights into the work that I was attempting.

5.1 Achievements

All of the initial goals were successfully attained and have resulted in a framework that can be used to distribute a plethora of applications. The end product allows developers of today to easily harness the power and scalability of the cloud, which otherwise could quite easily require laborious amounts of rewrite and programming man hours. Cirrus opens the doors for a whole host of scalable programs that would previously be reserved to only the most determined of developers. This result alone could deem this project a success.

In addition to the core goals, several extensions were also implemented which make Cirrus faster and more flexible to work with. Whilst a few of these were from my original proposal, major ones were conceived during the development stage and came about as a result of the agile development strategy employed.

From a personal point of view, Cirrus has furthered my understanding in a host of different fields, whether it be writing Python code for the first time, managing and administering cloud clusters, or simply how to evaluate applications in a quantitative manner.

5.2 Lessons Learnt

The most important lesson that I learnt was to check-in regularly with interested parties before implementing a complex idea. On a few occasions, I have ended-up rewriting sections of code following suggestions or insights discussed at subsequent meetings.

Throughout this project, I also learnt lessons concerning devising a new API from scratch. I now know to try and keep an API as flexible as possible. Too many times library functions that I had implemented turned out to be too constraining once I eventually started writing test programs to leverage them.

An important lesson learnt after coding in C was to make sure I was as explicit as possible with type information when declaring functions and variables. In the beginning, not doing so resulted in several memory allocation errors that would cause the program to crash at randomly determined intervals making it almost impossible to debug. Another important lesson was to enable all possible compiler warnings before I start a C project to save time in the long run.

5.3 Future Work

Cirrus is considered complete such that it can be already used to distribute many types of applications and it meets all of the success criteria originally detailed in the project's proposal. Ultimately, I hope to tidy up the code-base and merge the changes in to the main trunk of CIEL. This will make Cirrus available to anyone who wants to use the CIEL framework, but in the mean time it remains publicly available as a GitHub branch.

However, there is still plenty of scope for additional work. Some possible areas for future development are:

- **Introduce more effecient process checkpointing**

  As explained in section 4.4.1, the process checkpointing operation takes up the majority of the overall time taken to spawn a Cloud Thread. This could be improved upon, either by looking at other checkpointing solutions that may be faster under these circumstances, or by implementing a checkpoint cache (perhaps as part of the memoisation feature) where similar threads share the same checkpoint file.

- **Intorduce a more advanced heap implementation**

  The heap employed by the Shared Memory part of Cirrus was intentionally primitive as this was not the main focus of the project. New heap allocation routines that could better optimise memory when it is frequently freed and re-allocated would speed up this part of the framework.

- **Harness newer CIEL features**

  Recent CIEL revisions include several advanced features that could be leveraged by Cirrus to provide an even more seamless cloud experience. Examples include *tombstone references* (used to mark when data is no longer available) or *sweetheart references* (an annotated reference that specifies a preference to a particular worker node).

Bibliography

[1] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. *Computing Research Repository*, 2010.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI ' 04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA*, 2004.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[4] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch. The utility coprocessor: massively parallel computation from the coffee shop. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 7–7, Berkeley, CA, USA, 2010. USENIX Association.

[5] Donald E. Eastlake and Paul E. Jones. US secure hash algorithm 1 (SHA1). Internet RFC 3174, September 2001.

[6] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In Salim Hariri and Kate Keahey, editors, *HPDC*, pages 810–818. ACM, 2010.

[7] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[8] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Proceedings of SciDAC. 2006*, page 2006. Available: http://stacks.iop.org/17426596/46/494, 2006.

[9] Jim Highsmith and Martin Fowler. The agile manifesto. *Software Development Magazine*, 9(8):29–30, 2001.

[10] IEEE. *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995.

[11] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, June 2007.

[12] ISO. Iso c standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[13] Renaud Lottiaux, Pascal Gallard, Geoffroy Vallée, Christine Morin, and B. Boissinot. Openmosix, openssi and kerrighed: a comparative study. In *CCGRID*, pages 1016–1023. IEEE Computer Society, 2005.

[14] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, page 48, New York, NY, USA, 2009. ACM.

[15] Derek Gordon Murray and Steven Hand. Scripting the cloud with Skywriting. In *HotCloud '10: Proceedings of the Second Workshop on Hot Topics in Cloud Computing*, Berkeley, CA, USA, 2010. USENIX.

[16] Derek Gordon Murray and Steven Hand. CIEL: a universal execution engine for distributed data-flow computing. In *NSDI '11: Proceedings of the eighth symposium on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2011. USENIX.

[17] Russell Power and Jinyang Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–14, Berkeley, CA, USA, 2010. USENIX Association.

[18] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[19] Dimitri van Heesch et al. Doxygen, 1997.

Appendix A

Notes

A.1 Summation of Fibonnaci Series

In this chapter, I prove the following equation:

$$\forall n \in \mathbb{N}. \qquad \sum_{m=1}^{n} Fib(m) = Fib(n+2) - 1$$

Proof by Induction. First, let:

$$G(n) = \sum_{m=1}^{n} Fib(m) \qquad \text{and} \qquad H(n) = Fib(n+2) - 1$$

$$Fib(n) = \begin{cases} Fib(n-1) + Fib(n-2) & n \geq 2 \\ 1 & n = 1, 0 \end{cases}$$

Base Case

First, proof for the case where $n = 1$:

$$\begin{aligned} G(1) &= \sum_{m=1}^{1} Fib(m) \\ &= Fib(1) = 1 \end{aligned} \qquad \begin{aligned} H(1) &= Fib(1+2) - 1 \\ &= Fib(3) - 1 \\ &= 2 - 1 \\ &= G(1) \end{aligned}$$

Iterative Step

Second, proof that $G(n+1) = H(n+1)$ under the assumption that $G(n) = H(n)$:

$$\begin{aligned} G(n+1) &= \sum_{m=1}^{n+1} Fib(m) \\ &= Fib(n+1) + G(n) \\ &= Fib(n+1) + H(n) \\ &= Fib(n+1) + [Fib(n+2) - 1] \\ &= Fib(n+3) - 1 \\ &= H(n+1) \end{aligned}$$

□

A.2 Computer Specifications

For comparison purposes, the hardware specifications of the computers that I used to evaluate Cirrus are given below. It should be noted that both machines were running an OS under virtualization technology.

Development Virtual Machine

- Ubuntu 10.04 Lucid Lynx (32-bit)
  Parallels Desktop 6 VM under Mac OS X 10.6.7
- Intel Core i7 2.66GHz Dual-Core CPU
  256KiB L2 Cache, 4MiB Shared L3 Cache
- 8GiB RAM
  1067 MHz DDR3
- 500GiB HDD
  SATA 7200rpm
- NVIDIA GeForce GT 330M
  512MiB GDDR3 RAM

Amazon EC2 `c1.medium` Instance

- Ubuntu 10.04 Lucid Lynx (32-bit)
  Running under Xen Hypervisor
- 2 $\times$ 2.5 EC2 Compute Unit Virtual Cores
  Equivalent to a 2.0-2.4GHz dual-core Intel Xeon or AMD Opteron CPU. [1]
- 1.7GiB RAM
- 8GiB Elastic Block Store (EBS)
  Network mounted

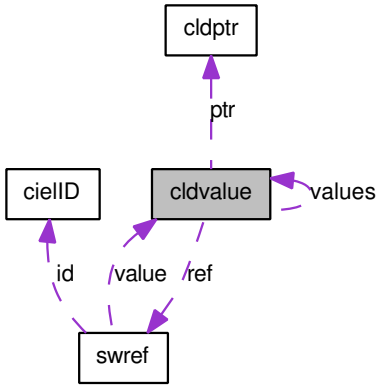
[1] One EC2 Computer Unit is stated as being equal to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor (http://aws.amazon.com/ec2/instance-types/).

Appendix B

Extract of Cirrus API Documentation

3.4 cldvalue Struct Reference

Collaboration diagram for cldvalue:



Public Types

- enum {
  NONE = 0, **INTEGER**, **REAL**, **STRING**,
  CLDPTR, **CLDREF**, **ARRAY** }

Data Fields

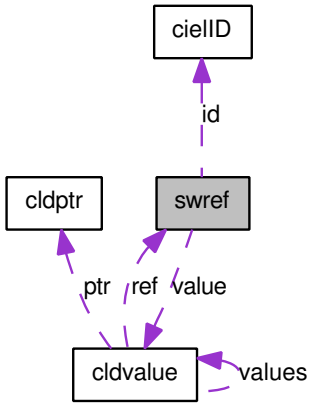
- enum cldvalue:: { ... } **type**
- union {
  intmax_t **integer**
  long double **real**
  char * **string**
  cldptr **ptr**
  struct swref * **ref**
  struct {
  struct cldvalue ** **values**
  size_t **size**
  } **array**
  } **value**

The documentation for this struct was generated from the following file:

- cldvalue.h

3.5 swref Struct Reference

Collaboration diagram for swref:



Public Types

- enum **swref_type** {
  SWREFTYPE_ENUMMIN = 0, **ERROR** = SWREFTYPE_ENUMMIN, **FUTURE** = 1, **CONCRETE** = 2,
  STREAMING = 3, **TOMBSTONE** = 4, **FETCH** = 5, **URL** = 6,
  DATA = 7, **OTHER**, **SWREFTYPE_ENUMMAX** = OTHER }

Data Fields

- enum swref::swref_type **type**
- cielID * **id**
- uintmax_t **size**
- char ** **loc_hints**
- size_t **loc_hints_size**
- struct cldvalue * **value**

The documentation for this struct was generated from the following file:

- swref.h

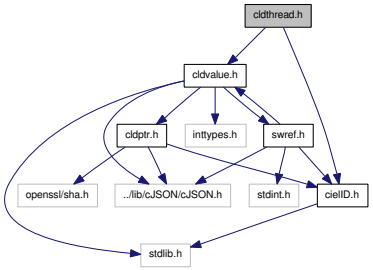
Chapter 4

File Documentation

4.1 cldthread.h File Reference

```
#include "cielID.h"
#include "cldvalue.h"
```

Include dependency graph for cldthread.h:



Defines

- #define cldthread_create(fptr, arg) cldthread_smart_create(NULL, fptr, arg)
- #define cldthread_current_thread() ((cldthread *)_ciel_get_task_id())
- #define cldthread_join(thread) cldthread_joins(&(thread), 1)
- #define cldthread_joins(threads, count) cielID_read_streams(threads, count)
- #define cldthread_close_streaming_result(thread) cielID_publish_stream(thread)
- #define cldthread_result_as_fd(thread) cielID_read_stream(thread)
- #define cldthread_free(thread) cielID_free(thread)

Typedefs

- typedef cielID **cldthread**

Functions

- int cldthread_init (void)
- cldthread * cldthread_posix_create (void *(*fptr)(void *), void *arg)
- cldthread * cldthread_smart_create (char *group_id, cldvalue *(*fptr)(void *), void *arg)
- int cldthread_stream_result (void)
- int cldthread_write_result (void)
- cldvalue * cldthread_result_as_cldvalue (cldthread *thread)
- swref * cldthread_result_as_ref (cldthread *thread)
- int cldapp_exit (cldvalue *exit_value)

4.1.1 Detailed Description

Author

Sebastian Hollington

Version

0.0.1

4.1.2 LICENSE

4.1.3 DESCRIPTION

This header file delares the API used by libCloudThreads.

4.1.4 Define Documentation

4.1.4.1 #define cldthread_close_streaming_result(thread) cielID_publish_stream(thread)

Finalise a thread's output stream and close the concerned file descriptor.

Returns

A non-zero value if the output was published (for the first time).

4.1.4.2 #define cldthread_create(fptr, arg) cldthread_smart_create(NULL, fptr, arg)

Create a thread for execution.

Parameters

← ***fptr*** Function that becomes the entry point of the new thread. The return value of this function becomes the value of the thread output (unless the output has been opened as a stream).

← ***arg*** Argument that fptr is executed with.

Returns

A value that represents the new cloud thread.

4.1.4.3 #define cldthread_current_thread() ((cldthread *)_ciel_get_task_id())

Retrieves a value that is used to represent the current cloud thread.

Returns

A value that can be used to represent the current cloud thread.

4.1.4.4 #define cldthread_free(thread) cielID_free(thread)

Free any memory associated with a thread pointer. Note that this is a local change i.e. it doesn't perform any operations on the cloud.

Parameters

← ***thread*** A pointer to the thread whose memory is to be freed.

See also

cldthread_create(), cldthread_smart_create(), cldthread_posix_create()

4.1.4.5 #define cldthread_join(thread) cldthread_joins(&(thread), 1)

Wait on the result of a thread.

Parameters

← ***thread*** A pointer to the thread concerned.

Returns

A non-zero value if a result was successfully retrieved.

4.1.4.6 #define cldthread_joins(threads, count) cielID_read_streams(threads, count)

Wait on the result of multiple threads.

Parameters

← ***threads*** An array of pointers to the threads whose result we want to evaluate.

← ***count*** The number of thread pointers in the array.

Returns

The number of thread values that were successfully retrieved.

4.1.4.7 #define cldthread_result_as_fd(thread) cielID_read_stream(thread)

Retrieve the result of a thread as a read-only file descriptor. This should be used to access data that is being streamed out by a thread that has called cldthread_open_result_as_stream().

The caller is responsible for closing this FD once they have finished with it.

Parameters

← ***thread*** A pointer to the thread concerned.

See also

cldthread_result_as_ref(), cldthread_result_as_cldvalue()

Returns

A file descriptor that we can stream thread output to.

4.1.5 Function Documentation

4.1.5.1 int cldapp_exit (cldvalue * *exit_value*)

Publish the global result for an application that is run in a cloud environment. The function returns EXIT_-SUCCESS for convenience.

Example:

```
int main(){
    ...
    return cldapp_exit( cldvalue_integer( 42 ) );
}
```

Parameters

← ***exit_value*** A pointer to the cloud value that will become the application's output.

See also

cldthread_result_as_cldvalue(), cldthread_result_as_fd()

Returns

EXIT_SUCCESS

4.1.5.2 int cldthread_init (void)

Initializes the libCloudThread framework.

Returns

A non-zero value if successful.

4.1.5.3 cldthread* cldthread_posix_create (void *(*)(void *) *fptr*, void * *arg*)

Create a thread for execution.

Parameters

← ***fptr*** Function that becomes the entry point of the new thread. The return value of this function becomes the value of the thread output (unless the output has been opened as a stream).

← ***arg*** Argument that fptr is executed with.

Returns

A value that represent the new cloud thread.

4.1.5.4 cldvalue* cldthread_result_as_cldvalue (cldthread * *thread*)

Retrieve the result of a thread as a cloud value.

The caller is responsible for freeing this value once they have finished with it.

Parameters

← ***thread*** A pointer to the thread concerned.

See also

cldthread_result_as_ref(), cldthread_result_as_fd()

Returns

A pointer to a cloud value.

4.1.5.5 swref* cldthread_result_as_ref (cldthread * *thread*)

Retrieve the result of a thread as a cloud reference.

The caller is responsible for freeing this reference once they have finished with it.

Parameters

← ***thread*** A pointer to the thread concerned.

See also

cldthread_result_as_cldvalue(), cldthread_result_as_fd()

Returns

A pointer to a cloud reference.

4.1.5.6 cldthread* cldthread_smart_create (char * *group_id*, cldvalue *(*)(void *) *fptr*, void * *arg*)

Start and schedule a thread for execution under memoisation. The parameters supplied are combined to create a unqiue identifier for the execution. If a thread with the same unique identifier has already been executed and produced an output then we won't bother to schedule the thread again, and return the previously computed result.

Parameters

← ***group_id*** A user-specified unique identifier that is used for memoisation.

← ***fptr*** Function that becomes the entry point of the new thread. The return value of this function becomes the value of the thread output (unless the output has been opened as a stream).

← ***arg*** Argument that fptr is executed with.

Returns

A value that represent the new cloud thread.

4.1.5.7 int cldthread_stream_result (void)

Stream output from the current thread. The returned file descriptor will close automatically after returning from the entry point of the thread or can be closed earlier using cldthread_close_result_stream(). Any thread attempting to join this thread will resume as soon as possible and can read from the stream using cldthread_result_as_fd().

Note: As soon as the thread returns, the stream to any reading Cloud Threads will be instantly terminated and they won't be able to read any further from it.

Returns

A file descriptor that we can stream thread output to.

Appendix C

Project Proposal

Sebastian Hollington
St. John's College
sh583

Part II Computer Science Project Proposal

Cloud Threading

Project Originator: Sebastian Hollington

Resources Required: See attached Project Resource Form

Project Supervisor: Malte Schwarzkopf and Derek G. Murray

Director of Studies: Dr. Robert Mullins

Overseers: Dr. Timothy G. Griffin and Dr. Markus Kuhn

Introduction and Description of the Work

Whether it be video-editing, music-encoding or matrix multiplications, many more applications are now parallelising their code to allow them to complete their computationally-intensive work loads more effectively.

Due to engineering challenges, processor clock-speed has plateaued in recent years and chip manufacturers are bringing multi-core and multi-threading CPUs to the consumer market. Most personal computers now ship with dual-core CPUs or CPUs which support simultaneous multi-threading technologies such as Intel's HyperThreading. But even still, it is extremely rare for PC hardware to support more than 8 concurrent threads. Spawning any additional software threads would bring little added benefit, as the operating system has to start sharing CPU time.

If there was a way, however, for an application to execute tens, hundreds (maybe even thousands) of threads concurrently, computationally intensive tasks could benefit from significant speed boosts which would previously have been unavailable to the consumer.

The Computer Laboratory's System Research Group (SRG) is currently developing a universal execution engine for distributed computing called "Ciel" [5, 6]. Ciel, like Google's MapReduce [2], aims to mask the complexity of distributed programming whilst providing a far less restrictive environment. In Ciel, a job is divided up into distributable tasks and lazy evaluation is used to schedule tasks to an available node.

POSIX is a set of IEEE standards which define a set of APIs for performing system and shell operations, e.g. process creation, file and directory operations, system timers. Since POSIX.1 was released in 1988, the standards have gone through several iterations bringing new standardised APIs, e.g. for shared memory and asynchronous I/O. POSIX.1c (IEEE 1003.1c [4]) brought new thread extensions to the framework, defining an interface for creating, scheduling, controlling and synchronising threads for concurrent execution. Implementations exist for popular operating systems, all of which spawn and manage system threads.

I propose to write a thread-management API (similar to POSIX.1c) that, instead of scheduling *system* threads, distributes and assigns these threads to remote nodes on a distributed network. This will be done by extending the current Ciel framework so that it can manage, package and distribute processes, process-state and any shared memory. The thread-management API will then interface with Ciel to spawn and manage thread-tasks on a distributed computer network. This could potentially make hundreds if not thousands of idle cores around the globe available to a single computer program, considerably reducing running-times for highly-parallelised tasks.

Resources Required

Development will predominantly be on my personal Unix and Windows 7-based computer system.

The university PWF facility will be used for back-up purposes in conjunction with the Student Run Computing Facility (SRCF) and regular snapshots of the project repository will be hosted using a private online back-up service.

As Ciel is written in the cross-platform Python language, extending the framework will no-doubt require the use of Python too.

To achieve maximum reach and transparency, I intend to write the thread-management API in C, which would also facilitate any future convergence with the POSIX.1c API (see *Variants* section below). I have worked with C and C++ libraries a lot in the past and have written APIs to wrap around C functions in previous freelance programming work.

Starting Point

To complete the project, I will draw from:

1. **Part 1B Courses from the Computer Science Tripos**

   Specifically, the *Concurrent and Distributed Systems* and *Programming C and C++* courses from Lent Term will be relevant.

2. **Previous Programming Experience**

   I will be building on programming experience gained during my software development internship over the summer and previous freelance programming assignments.

Substance and Structure of the Project

The objective of this project is to successfully design and implement a thread-management library in C that, instead of creating threads locally, schedules threads as remote tasks using the distributed Ciel framework.

The full POSIX.1c thread extensions library contains approximately 100 functions for performing various different thread-related operations. My aim is to implement a small working subset of these in the time available, focusing on those that will provide the most flexibility to programs and which are most relevant to the problem at hand.

To achieve this, various different hurdles will have to be overcome. Firstly a distribution mechanism for each thread's binary code will need to be devised, and corresponding support

will have to be added to the Ciel framework. This is particularly tricky as a thread's program code is typically specified using a function pointer, which, because of virtual addressing, is likely to be meaningless in any other execution environment. As such, integrating process-level checkpoint/restart (e.g. BLCR [3]) may be necessary. This could potentially restrict the operating system of master or worker nodes to the Unix platform, but further research needs to be done in this area first.

Mechanisms for initialising the thread, sharing state information and synchronising multiple threads will have to be carefully planned and designed in a way that would be attainable using the Ciel paradigm. Protocols will need to be devised, most likely using the same JSON communication channels currently employed by Ciel.

Another area that will require careful consideration is the sharing of data and main-memory between the execution contexts on the parent and the worker nodes. A lot of threads make frequent use of shared memory, which is quite a big topic in itself when concerned with distributed systems. To confine the problem space, the aim will be to provide all worker thread processes with read-only access to regions of predetermined memory using a `shrdmalloc()`command. Threads will communicate results using only their return values (akin to fork/join parallelism [1]). Time permitting, message-passing routines could be used to provide more flexible synchronised primitives such as futures or condition-variables (see *Optional Extensions* section below).

Variants

It may be the case that the proposed thread-management API converges with the API defined in POSIX.1c. In this case, enabling cloud-thread support in an application may become as simple as linking against the cloud-thread implementation of the POSIX.1c API.

Optional Extensions

The following work could be undertaken if found feasible as the project progresses:

1. Investigation and implementation of more flexible synchronisation primitives such as futures or condition-variables.
2. Forwarding and collection of I/O that is displayed or retrieved by the worker threads, to/from the master node.
3. Run-time fallback where it might be more efficient (or perhaps necessary in an offline-environment) to revert one or more of the threads to a typical OS-scheduled one.

Success Criteria

For the project to be deemed a success the following items must be successfully completed.

1. A plan for distributing a process amongst nodes (inc. program code, state and memory).
2. Implementation of the plan by extending the CIEL framework to accommodate, e.g. process-level checkpointing.
3. An implementation of an API that is flexible enough to schedule cloud, or potentially system threads, transparently to the programmer.
4. A set of parallel algorithms should be written and used to use in development; as test-cases; and in verifying that the solution is working.
5. Tests should be performed to determine the speed and scalability of the resulting solution.
6. The dissertation must be planned and written.

Timetable and Milestones

The project will be divided up into ten approximate three-week blocks. All work is intended to be completed by the start of May, just under three weeks before the final deadline of Friday 20th May 2011.

0. October 1st - October 21st

Preparation and submission of project proposal. This included finding and researching ideas and planning the project timescale.

1. October 25th - November 7th

Work on the project will begin. Further research will be done to select the available frameworks and external tools necessary to complete the task and how they will interoperate. One of two parallel algorithms will be written for testing and benchmarking purposes in future stages.

Milestone: Determination and finalisation of any plans, in particular the use of third-party research or frameworks.

2. November 8th - November 28th

The CIEL framework will be extended to be able to save, package and distribute a process running on the parent-node to worker-nodes. This has to be done first as the method employed will dictate the flexibility of any proposed API that will follow.

Milestone: Processes should be able to be "checkpointed" and executed on remote systems using the CIEL framework and bespoke testing scripts.

3. November 29th - December 19th

A basic API will be implemented for creating, joining and terminating threads, using the CIEL extension previously implemented. Simple thread-spawning test programs will be used for testing.

4. December 20th - January 9th

Design and implement an extension to allow CIEL to handle common shared memory blocks that would be returned by a `shrdmalloc()` command. This will probably involve extending the data table memory mechanism already present in CIEL.

5. January 10th - January 30th

Progress report will be written and the presentation formulated. The deadline for the progress report is Friday 4th February 2011.

6. January 31st - February 20th

Parallel algorithms will be used to rigorously test *and* benchmark the project.

Milestone: Parallel algorithms should be able to execute and return meaningful results.

7. February 21st - March 13th

Further extend the CIEL framework to support futures or condition-variables using message-passing techniques. Devise and implement a corresponding API. (Optional Extension)

8. March 14th - April 3rd

This period stretches outside of full-term, and will be used as buffer time for any previous stage(s) that may have overrun. Any available extra time can be used to carry out optional extension work.

Milestone: Finish all coding, implementation and testing.

9. April 4th - April 24th

Write the dissertation for the project and submit several iterations for review by supervisors. This will involve evaluating the project and collecting any data that can be presented.

10. April 25th - May 1st

Final changes should be made to the dissertation after receiving feedback. The dissertation should be complete by the start of May.

Any remaining time will primarily be used for exam revision, but may use it as buffer time if any previous stage overruns.

References

[1] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. *CoRR*, abs/1005.3450, 2010.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[3] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Proceedings of SciDAC. 2006*, page 2006. Available: http://stacks.iop.org/17426596/46/494, 2006.

[4] IEEE. *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995.

[5] Derek G. Murray and Steven Hand. Scripting the cloud with skywriting. In *HotCloud'10: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[6] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapedddy, and Steven Hand. Ciel: a universal execution engine for distributed data-flow computing. 2010. Under Submission.