David Adeboye

Parliament: A distributed general-purpose cluster-computing framework for OCaml

Computer Science Tripos – Part II

Jesus College

May 16, 2019

Declaration

I, David Adeboye of Jesus 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 **David Adeboye**

Date **16/05/2019**

I, David Adeboye of Jesus College, am content for my dissertation to be made available to the students and staff of the University.

Proforma

Candidate Number:	**2381B**
Project Title:	**Parliament: A distributed general-purpose cluster-computing framework for OCaml**
Examination:	**Computer Science Tripos – Part II, May 2019**
Word Count:	**11989**[1]
Line Count:	**7964**[2]
Project Originator:	Dr J. Crowcroft & The Project Author
Supervisor:	Dr J. Crowcroft

Original Aims of the Project

OCaml's language features make it very appealing for big data processing libraries. However, OCaml applications cannot run in parallel and are therefore limited from taking advantage of modern multicore processors. I hypothesise that is possible to achieve performance gains for an OCaml execution in typical big data processing workloads by distributed computation. My aim is to create a distributed library and scheduler, supporting the development of distributed OCaml applications. This will contribute to easier, more efficient use of computing clusters, where the scheduler can easily share computing power between multiple applications.

Work Completed

I successfully created both a distributed library for OCaml and a cluster scheduler which enables OCaml applications to parallelise over multiple cores. Extensive results proved that my project decreases running time of a OCaml application. Big data processing benchmarks showed a 81% speedup, with a large decrease in memory usage and minimal

[1]This word count was computed by `texcount -inc -sum -total -utf8 diss.tex` and includes headings, footnotes, tables and captions. Content outside chapters 1–5 is excluded using `%TC:ignore` and `%TC:endignore markers`, and elements such as tables included using flags such as `%TC:group table 1 1`, `%TC:group tabular 1 1` and `%TC:group tabularx 1 1`.

[2]This line count was computed by using the command `wc -l` on all written code. It includes unit and integration tests, Protocol buffer specification files & evaluation code, but does not include build system configuration files or autogenerated code.

CPU overhead. I also implemented both of my extensions, adding both job isolation and fault-tolerance to my system. Consequently, I met all of my project proposal's success criteria, proving the viability of OCaml as a choice language for big data processing.

Special Difficulties

None

Contents

List of Tables and Figures

Chapter 1

Introduction

This dissertation describes the implementation and evaluation of a cluster computing framework for increasing the performance of OCaml applications. The implementation is empirically shown to achieve over a 5-fold decrease in running time on a multicore machine on data processing workloads and is up to 10x faster than the Spark's scheduler (§1.3) at task assignment.

1.1 Motivation

OCaml is a functional, general-purpose programming language with an emphasis on safety and speed with several features that make it appealing for big data processing. [1] The strong type system means that the OCaml compiler can easily detect unhandled cases before compilation. This is a common problem in data processing, which typically isn't caught until — computationally expensive — sample runs. [2] Additionally, the mark & sweep garbage collector means OCaml avoids long pauses by not copying objects through generations, allowing it to easily handle large objects, a flaw found in most generational GCs. [1] [3]

However, the OCaml runtime executes in one kernel thread [4], preventing its applications from benefiting from multiple cores/processors. With new CPUs prioritising core count over clock speed [5], OCaml's performance is restricted, as it cannot use this increasing processing capacity found in these additional cores. Data processing applications written in OCaml are therefore at an disadvantage, compared with other popular languages with native support for parallelism.

Furthermore, as core counts rise, the physical limitations of the bus bandwidth mean that memory access latency is increasingly non-uniform, so locality-of-reference is progressively more important to maximise cache usage. [6]

1.2 Project Summary

I present `Parliament`, a framework focused on evaluating the viability of big data processing in OCaml, by distributing the computation across multiple cores by exploiting spatial locality-of-reference through data parallelism. `Parliament` provides a simple OCaml library for writing parallel applications, and a cluster scheduler that handles automatic task assignment with in-built support for fault-tolerance and isolation.

In this dissertation, I explore the hypothesis: it is possible to achieve performance gains for an OCaml execution in typical big data processing workloads by distributing them over multiple cores. Unlike other work in this area (§1.3), `Parliament` supports multiple users to share a processing cluster and defines a functional programming model that can be used by other languages. The multithreaded components of the system are constructed in Rust [7], which offers more predictable (high) performance than typical garbage collected languages, through an ownership-based memory model.

Extensive empirical evidence shows that `Parliament` takes advantage of multiple cores and significantly decreases the latency of typical workloads, supporting my hypothesis. It successfully handles failures of any component in the system, including cluster masters, and provides isolation from other running processes. `Parliament` takes a different approach to most cluster computing libraries (§1.3) and provides a low-level API allowing the developer to select an appropriate data structure format for the specific problem.

1.3 Related Work

Spark: Spark is a commercial, general-purpose cluster-computing framework [8]. Spark uses Resilient Distributed Datasets (RDDs) to provide a restricted form of shared memory. *Spark, however, is limited to JVM-based languages & requires you to use their data API, which can be limiting in particular use cases.*

Functory: Functory is a distributed computing library for functional programming languages [9]. Functory provides facilities for marshalling data and tasks between a set of always-running workers. *Nevertheless, this requires machines to be statically defined in the code & only one workload to use the defined cluster at any one time.*

JoCaml: JoCaml is an extension to OCaml for concurrent and distributed programming inspired by join-calculus [10]. The system uses the actor concurrency model to share state between threads and threads can be dynamically spawned. *The extension, however, has no native support for fault-tolerance.*

Chapter 2

Preparation

Having established my project's hypothesis, I performed some work prior to implementing my system. In this chapter, I describe my starting point, introduce background knowledge, explain language choices, perform some requirements' analysis, establish my library interface requirements and finally plan my engineering practices.

2.1 Starting Point

Upon starting this project, my experience was as follows:

- **OCaml:** My knowledge of OCaml was limited to studying a similar language: Standard ML from Part 1A Foundations of Computer Science [11], however, I had no experience writing/structuring a large system in the language.
- **Rust:** I had no prior experience programming in Rust, this project was my first exposure to the language, and languages without a garbage collector or explicit pointers.
- **Distributed Computing Systems:** I had limited exposure to batch processing systems through personal projects and summer internships.
- **Git:** I was skilled in Git through Part 1B Further Java [11], group project and internships; however, I had limited experience in using and setting up continuous integration systems.

2.2 OCaml Multi-processing

OCaml programs cannot run in parallel across multiple cores [4]. The main opposition being the garbage collector, which pauses execution removing any need for data barriers when performing clean up operations. However, CPUs are increasing in core count [5] and if applications want to take advantage of the additional capacity across cores, they are required to become more multithreaded. This problem is exacerbated by technologies such as Intel's Hyper-Threading [12], which introduces two "virtual" cores per physical

core, interleaving instructions, to execute two threads concurrently, further prioritising multiple threads over single core performance.
The lack of support for parallelism also limits the development of distributed computing libraries for OCaml; these libraries need constant communication with other processes for both monitoring and message passing. These operations, which are easily parallelised, waste single-threaded performance on communication, reducing the efficiency of the system. OCaml offers limited native support for multi-processing, however, it is quite crude, involving forking of processes rather than providing any multicore abstractions. Due to this limitation, I opted to build several parts of the system in the Rust programming language.

2.3 Rust

I chose the Rust programming language to implement the highly parallel components of the system. [7]
Instead of a garbage collector or manual memory management, Rust relies on an ownership model where data is *owned* by a thread, and once all pointers to a variable are unreachable, it is automatically freed. [13] This allows Rust, at the compile stage, to determine exactly when data will be freed and place directives to free the memory location. Memory management is a big concern in data processing frameworks. For example, choice of the garbage collector in Apache Spark can increase performance 3-fold. [14]
Choosing Rust also shifts a significant amount of the memory safety checking to the compilation stage. For example, Java, during execution, checks whether a thread has made an illegal change to a memory location, throwing an exception if it does. In comparison, Rust's strict borrow checker can detect this scenario and abort the compilation, minimising the number of run-time checks. The Rust compiler also informs the developer about any common concurrency pitfalls that exist in the code, aiding the development and iteration of a highly concurrent system. [13]

2.4 Parallelism

Multicore CPUs introduce parallelism into computer systems, allowing programs to execute more instructions per second than previously possible, giving applications several fold more performance. However, it does not automatically give linear speedups as one might expect. Amdahl's law [15] gives the maximum speedup in latency of the execution of a task. It shows potential speedups are primarily limited by the sequential proportion of the problem:

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

where p is the proportion of execution time that can be parallelised and s is the speedup of parallelising that proportion

This leads to an upper bound of the relative speedup of a fixed task as compared to execution on a single processor.

$$\lim_{s \to \infty} S_{\text{latency}}(s) = \frac{1}{1-p}$$

Therefore, to exploit these potential gains, a significant portion of work in parallel computing is focused around decreasing the sequential proportion.

2.5 Cluster Computing

To achieve parallelism past a single machine, we can use cluster computing to increase performance. This involves using multiple machines to achieve a single purpose. These computers work together in the execution of computationally intensive workloads which would otherwise not be possible using a single computer, in a useful timeframe. The main advantages of such a system include high availability/resilience to failures: a cluster is not dependent on any single machine: problems with individual machines do not halt the progress of the entire cluster. This, however, leads to a growth in the cost of building/maintaining infrastructure due to the increased complexity and sheer number of machines.

2.5.1 General Architecture

In cluster computing, nodes fall under one of two categories: masters and workers.

Master: These provide control of the worker nodes and take any task (user code) submissions. They allocate tasks to worker nodes, scheduling and monitoring, but rarely execute any of the tasks.

Worker: These provide an environment for running tasks assigned by the master node(s). As demand increases, worker nodes can be added to perform more computation.

Typically, a cluster contains significantly more worker nodes than master nodes.

2.5.2 Data-parallel computations

Data parallelism is a model of performing parallel execution on multiple processors by distributing data across different nodes, popularised by MapReduce [16]. Instead of providing a shared memory model, where each thread has (or appears to have) direct access to the entire dataset, users create a directed acyclic graph of stages, where each stage will perform the same operation, in parallel, to different subsets of the dataset. These operations typically contain no side effects. This allows them to be easily scaled, and rescheduled in the case of failures, with no effect to the final value, in a manner opaque to the user. Data parallelism encourages sequential reads of data, allowing the hardware to take advantage of spatial locality-of-reference. Spatial locality reduces the number of cache misses, increasing processor throughput by reducing the number of stalls. [17]

This model also lends itself to sequential languages, where each of the stages can be (sequential) executions, run on different partitions of the data in parallel, and be composed together as a distributed execution. In distributed search (see Figure 2.1), each *Search* and *Collect* task can be run sequentially, and the parallel execution is gained by running multiple tasks at the same time, over a different subset of the file.

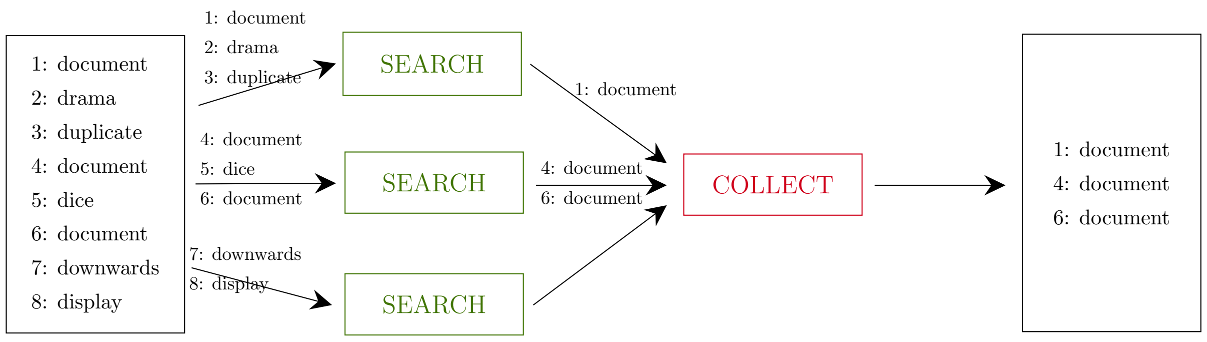


Figure 2.1: *Overview of an example MapReduce workload of a distributed search*

2.5.3 Cluster Computing goals

Apache Mesos Essentials [18] states 5 competing goals cluster managers aim to achieve:

1. **Efficiency:** The main aim is to achieve as close to 100% of your compute resources on *useful* work, i.e. workloads.

2. **Isolation:** Clusters are usually shared between many workloads; therefore, interactions between tasks could lead to unwanted side-effects.

3. **Scalability:** Clusters are always continually growing; therefore, the ability to take advantage of ever-growing compute resources is important.

4. **Robustness:** Due to the nature of scale, there is likely to be regular machine or network failures; therefore, it is important that clusters are resilient to such.

5. **Extensible:** During operation, changes in workloads and hardware invariably will require an adaption to properly ensure that the cluster is being used efficiently.

Typically, optimality in a particular goal can only be achieved by making trade-offs in the others. In preparation for building `Parliament`, I chose isolation and robustness as my most important priorities to provide, as they can be easily qualitatively and quantitatively verified. Goals such as efficiency and extensibility are excellent goals to aim for, however, there is a wide range of different approaches and it made sense to aim for more complete goals, than seemingly endless optimisations for a Part II project.
Even though Rust offers performance improvements over typical garbage collected languages, techniques such as Shuffle File Consolidation [19], can lead to performance gains of over 200% on typical Map-Reduce workloads. These approaches also typically require a high-level data abstraction where the cluster dictates the organisation and types of the data being exchanged.

2.6 Fault-Tolerance

One of the main desirable assurances for `Parliament` is the ability to recover from any faulty component. The main type of failures to be considered are fail-stop failures, exhibited as a node crash, typically due to a networking fault, delay, or process crash on the node. To ensure fault-tolerance, we must detect faults both in worker and master nodes.

2.6.1 Worker Fault-Tolerance

Worker management is typically centralised in a cluster, making fail-stop failures easy to detect in workers. They are typically detected using periodic polling of workers (i.e. heartbeat) to detect whether they are still working in a correct manner.

2.6.2 Master Fault-Tolerance

Master node management, on the other hand, is not centralised, requiring complex distributed algorithms. Aside from fault-tolerance, another advantage is the ability to host clusters in different physical sites, each with their own replicated master, which is helpful if the system offers availability guarantees. The main trade-off involved is other competing goals: efficiency and scalability. Due to the added network and coordination overhead, there is likely to be a decrease in performance due to an increase in computation and request latency. Other trade-offs include potential reductions in consistency and increased complexity. What we are trying to achieve is *Replica Coordination*: where a set of servers jointly, agree on a final decision so they can make progress. One method of achieving this is by using Replicated State Machines.

Replicated State Machines

Replicated State Machines [20] are used to implement fault-tolerant services by replicating servers and coordinating client interactions with these replicas. By modelling a system as a deterministic state machine, instructions can be duplicated and sent to several replicas of such a system in the same order. This should cause the same state transitions and outputs. This is a useful concept as multiple masters support can be added to almost any system — which can modelled as a state machine — by replicating (ordered) client requests and monitoring the servers to ensure they are in the same (consistent) state.

The key difference between workers and masters is the importance of their state. If a worker fails, a master can simply reallocate its job to another worker. If a master fails, the state needs to be recovered. RSM ensures that cluster management state is durable.

2.7 Isolation

Another desirable assurance for `Parliament` is ensuring tasks are isolated from each other. The main driving factor is accidental interactions with other tasks, i.e. editing other tasks' intermediate files.

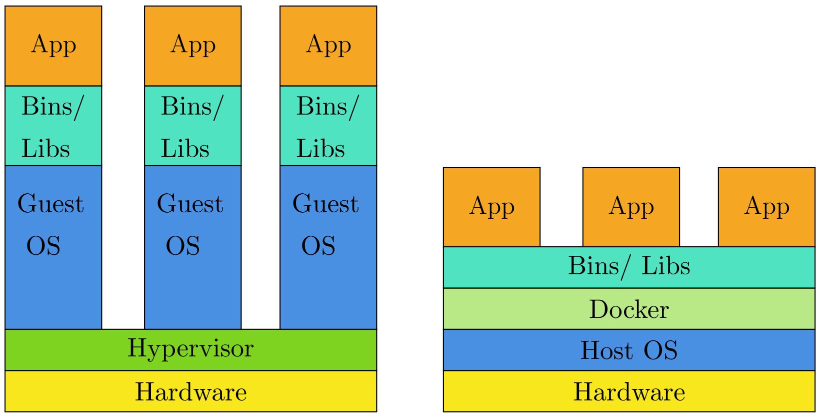


Figure 2.2: *Comparison of virtual machines & containers. Each VM requires the full overhead of an OS. Containers instead share a kernel*

Docker [21] is a program that handles a method of isolation known as *containerisation*. Instead of running a hypervisor to hold multiple virtual machines, containers sit above the operating systems, as a light-weight wrapper around an executable. Containers run in the OS (see Figure 2.2), allowing them to share binaries and libraries with other containers, saving disk space and memory, something not possible with virtual machines.
Docker offers the ability to create these isolation units, easily without incurring the set-up performance hits of a VM. Docker containers are built by specifying a "plan" named *Dockerfiles*, a set of instructions of how to install and run the program executable. These *Dockerfiles* are used to create an image, which are templates for containers. These images are usually pushed to a central registry and then subsequently pulled onto other machines. When an image is run, Docker adds necessary elements such as a filesystem and networking elements for the template and creates a container.
Research into container usage in high performance clusters shows that under typical workloads, they incur a 2.21% mean loss in performance [22]. This shows Docker is an efficient tool for providing isolation without a large performance hit. However, even a 2% loss in performance may be unacceptable in a large cluster, as it could require 100s of additional machines.

2.8 Requirements Analysis

I have outlined the main requirements of the project (see Table 2.3). My project's design naturally provided a separation between units, outlined as different requirements. The deliverables marked High are required for the **Core** of the project, while Medium and Low were desirable **Extensions** to the project but were not required for the success criteria.

2.9 Professional Practice

Designing distributed systems can be quite complicated, especially as I have no experience in system development of this scale. Throughout the project, I ensured I was following practices laid out by similar systems. Extra care was taken, as a goal of this project was

	Requirement	Priority	Difficulty
Core	OCaml user library	High	Medium
Core	OCaml worker library	High	Low
Core	Master implementation	High	High
Core	Worker implementation	High	Low
Core	Sample distributed workloads	High	Low
Extension	Add containers support	Medium	Medium
Extension	Add multiple masters support	Low	High

Table 2.3: *Summary of deliverables*

to be a library, therefore not only must it be functioning but it must also provide any potential users with adequate documentation.

2.9.1 Testing

Once I planned the overall system design and the communication between the systems, I adopted a test-driven approach for creating the library. The testing and verification of distributed systems is complex [23] and so I took a lot of care to ensure assurances could be verified.
To perform testing I used the established 3 testing techniques presented in *Testing Distributed Ada Programs* [24], in addition to standard unit tests to verify my applications:

1. **Test the internal logic of a unit on the host, using dummy units to send/receive messages to/from the unit of interest:** Mock testing allows developers to test specific components' behaviour when interacting with others by simulating real components with *mock* objects. I employed mocking to ensure the communication in/out of a module adhered to the strict standard I had placed. This also allowed me to test each component independently, which supported swift development of the individual modules.

2. **Test the unit when it interacts with the real versions of other units, still on the host with units executing with pseudo parallelism:** I added integration tests to my continuous integration system which would test the latest components against each other, I achieved pseudo parallelism by setting the CPU core affinity.

3. **Test the units when they are distributed (between the host and target(s) or between targets), i.e. executing with true parallelism:** Upon completion of my project, I ran sample workloads on my implementation to verify it was performing the correct computation.

2.9.2 Licensing

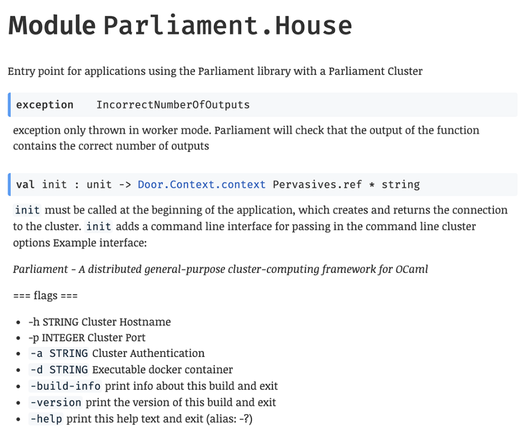
I ensured that my work conformed with all relevant licences.

- **BSD Licence:** Protocol Buffers
- **MIT License:** Jane Street Core, The Rust Project, ByteOrder, Chashmap, Clap, Crossbeam, Tokio, Shiplift

These licences permit me to publish my project as an open-source library.

2.9.3 Workflow and Tooling

Having no prior experience in my two chosen languages, to ensure I wrote readable and maintainable code, I adhered to code standards provided by Jane Street for OCaml open source [25] and the official Rust documentation [26] for OCaml and Rust code.
Besides creating Git repositories for each module, I used the Travis continuous integration system [27] which ensured all code pushed passed to all unit and integration tests and conformed to code standards. This also ensured any results were reproducible in a controlled manner.
Finally, I created an `odoc` (OCaml documentation) for the OCaml library (see Figure §2.4), providing an easy entrance for the library for any users. I also created `man` pages for the Rust executables (see Figure 2.5), — UNIX's preferred system for documentation — and summaries for the command-line options.



Module `Parliament.House`

Entry point for applications using the Parliament library with a Parliament Cluster

```
exception   IncorrectNumberOfOutputs
```

exception only thrown in worker mode. Parliament will check that the output of the function contains the correct number of outputs

```
val init : unit -> Door.Context.context Pervasives.ref * string
```

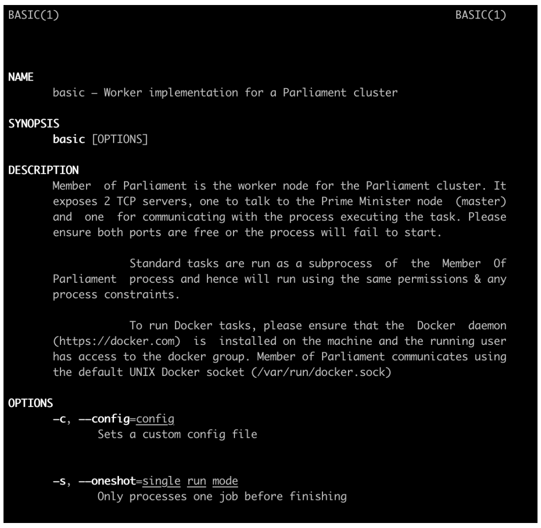
`init` must be called at the beginning of the application, which creates and returns the connection to the cluster. `init` adds a command line interface for passing in the command line cluster options Example interface:

Parliament - A distributed general-purpose cluster-computing framework for OCaml

=== flags ===

- -h STRING Cluster Hostname
- -p INTEGER Cluster Port
- `-a STRING` Cluster Authentication
- `-d STRING` Executable docker container
- `-build-info` print info about this build and exit
- `-version` print the version of this build and exit
- `-help` print this help text and exit (alias: -?)

Figure 2.4: *ODoc for the OCaml Parliament library*



```
BASIC(1)                                                          BASIC(1)



NAME
       basic - Worker implementation for a Parliament cluster

SYNOPSIS
       basic [OPTIONS]

DESCRIPTION
       Member  of Parliament is the worker node for the Parliament cluster. It
       exposes 2 TCP servers, one to talk to the Prime Minister node  (master)
       and  one  for communicating with the process executing the task. Please
       ensure both ports are free or the process will fail to start.

                 Standard tasks are run as a subprocess  of  the  Member  Of
       Parliament  process and hence will run using the same permissions & any
       process constraints.

                 To run Docker tasks, please ensure that the  Docker  daemon
       (https://docker.com)  is  installed on the machine and the running user
       has access to the docker group. Member of Parliament communicates using
       the default UNIX Docker socket (/var/run/docker.sock)

OPTIONS
       -c, --config=config
              Sets a custom config file


       -s, --oneshot=single run mode
              Only processes one job before finishing
```

Figure 2.5: *`man` page for Member Of Parliament module*

Chapter 3

Implementation

In this chapter I describe the implementation of `Parliament`. It proved to be a substantial undertaking with the final system, excluding evaluation code, totaling over 7,500 lines of code. Given the breadth of the system, I will largely discuss the high-level design decisions made, and the modules that proved challenging in both development and debugging.
I will first present the conceptual model of the library, then the architectural design of each module. Then, I will discuss in detail the implementation of core parts of the library and the extensions I have made. For brevity, I will not go into detail about the low-level technical details unless important.
Both extensions were designed and implemented to be independent, optional add-ons to `Parliament`, which not only makes evaluation easier, but gives the users of the library flexibility on what trade-offs they can make.

3.1 Functional Approach

`Parliament` is a data-parallel distributed library, where the architecture encourages developers to write parallelised workloads, while not restricting developers from writing whatever function they want. I avoided a high-level data abstraction such as Spark's Resilient Distributed Datasets [8], and instead moved the responsibility of data partitioning to the developer, allowing them to optimise based on data statistics.

To submit to the processing cluster, developers submit an (ordered) list of jobs (named a workload). Once the workload has successfully been executed, the resulting output value is returned to the user. These jobs are simply OCaml closures defined in the user's application. The functional environment allowed me to be generic, allowing jobs that implement any of the following polymorphic functions:

- **Single-In-Multi-Out:** $\alpha \to \beta$ `list`
- **Single-In-Single-Out:** $\alpha \to \beta$
- **Multi-In-Single-Out:** α `list` $\to \beta$

These represent the cardinality of the data values handled by the jobs, i.e. **Single-In-Multi-Out** denotes a job that requires one input value and generates multiple output values. A similar format follows for the other two types. *The number of output values does not need to be known at job submission time.*

*A **Multi-In Multi-Out** type is not used, as in the situation of multiple input values, the cluster would not know whether to create one or several tasks.*

A task is an instance of a job running on a particular piece of data. Once the input data is available, the cluster creates these tasks and iteratively submits them to workers. The number of tasks generated for a particular job by the cluster is solely dependent on the number of outputs (i.e. pieces of data) produced by the previous job and job type.

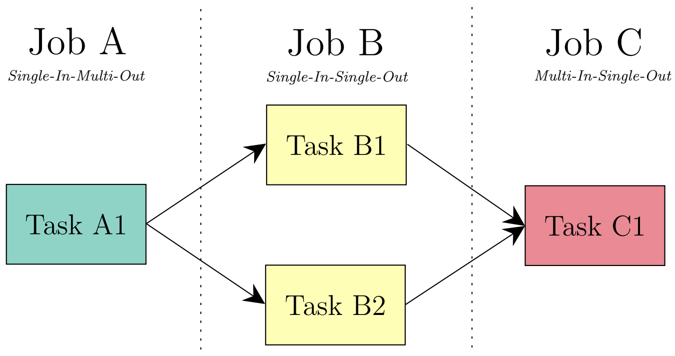


Figure 3.1: *Example data flow between tasks in jobs*

Taking the example given in Figure 3.1, after executing task A1, the cluster creates two tasks for job B, as the task in job A returned two values. Once the **Single-In-Single-Out** tasks in job B complete, the singular task in job C takes both of the results of job B as its input: **Multi-In-Single-Out**.
In the case of an error, or exception in a task, the workload is aborted, and all running tasks in the same job and any subsequent jobs are cancelled. For example, if task B1 fails, task B2 would be pre-empted and job C cancelled. A successful execution requires all tasks in all jobs to execute without errors.

3.1.1 Datapack

OCaml's static type checker requires the compiler to know (and verify) the types of all functions and data. This means `Parliament` is required to *know* the type of each users' closures, in order to invoke them, which is not possible at compilation.
To solve this problem, I created an OCaml datatype, which allows `Parliament` to know the types of all jobs submitted to the cluster. *datapack* is a wrapper around a byte array (see Listing 3.2), used to store the marshaled values of the user's data. Taking the example shown in Figure 3.1, the datapack returned by job A would contain two values, while the datapacks returned by each task in job B would each contain a single value.
In `Parliament`, every job has the type [*datapack* → *datapack*], so that the library can determinably invoke jobs' function closure. Taking the definition shown in Listing 3.2,

[0]Libraries are typically compiled before user code is written

```
1  type datapack = {
2      data : bytes array;
3  }
4  val create : int -> datapack
5  val add_item :  datapack -> 'a -> int -> unit
6  val get_item : datapack -> int -> 'a
7
8  (* Example usage
9  let map_function file dp =
10     let lines : string list = Datapack.get_item dp 0 in
11     ...
12     let datapack = Datapack.create 1 in
13     Datapack.add_item datapack tbl 0;
14     datapack *)
```

Listing 3.2: *Truncated version of* `datapack.mli` *with example usage*

developers define the type of their closure (as [*datapack* → *datapack*]) then extract values from the datapack. On transmission, this data is then converted to bytes using OCaml's Marshal library [28]. Marshalling involves transforming an object's representation to a format that can be easily transmitted. However, native marshalling is not type-safe [28], therefore, it's the responsibility of the developer to ensure that values' types are correct (see Line 10 in Listing 3.2).

3.2 System Design

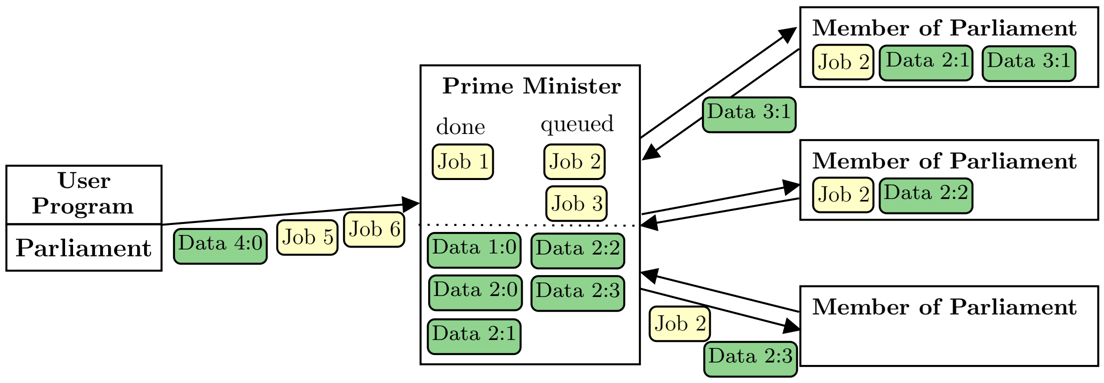


Figure 3.3: *High-level system design of* `Parliament`

There are 3 main parts to this system:

1. *Prime Minister:* Cluster master, Rust executable that communicates between the users and workers, handles fault-tolerance and task assignment.

2. *Parliament:* OCaml library, which handles data marshalling and communication with the cluster.

3. *Member of Parliament:* Cluster worker, Rust executable that communicates with the Prime Minister, that wraps around the OCaml executable and runs closures in the OCaml executable.

3.3 Prime Minister

Prime Minister, aptly named, is the master node of cluster taking workload requests from users and assigning them to workers. The main two parts of *Prime Minister* are its data structures, holding the cluster state, and the threads that operate on them.

3.3.1 Data Structures

The data structures in *Prime Minister* were influenced by the large amount of parallelism desired. Deadlocks occur when a waiting thread is still holding on to another resource that the first needs before it can finish, leaving a system in an undesired state causing progress to halt. These structures were chosen to minimise lock contention on data and the chance of a deadlock.

The majority of data stored in *Prime Minister* is stored in 4 concurrent hash maps[1], for users, jobs, tasks and data, mapping between their respective IDs and their structs. Concurrent hash maps were chosen because they provided separate writer and reader locks, which is important for maintaining a large read rate (status updates) but still supporting infrequent writes (finished tasks).
Prime Minister also contains two lock-free queues [29], a job queue and a task queue. The job queue contains available-to-run jobs that have not yet been split into tasks. The task queue contains waiting-to-run tasks that have not yet been assigned to a worker. Both queues are used to implement a simple work-conserving first-in, first-out (FIFO) scheduler.

The job queue does not contain jobs that are blocked, i.e. jobs that are waiting for previous jobs to complete. A naïve approach would be to create a set of all blocked jobs and periodically check whether any jobs are eligible to run, however, this would lead to an $O(n)$ implementation. This would increase the latency of the MCT, reducing its performance in metrics such as task assignment.
Since, on submission, we know the exact order of the jobs, a better approach, one which `Parliament` uses, is for each job struct to contain the **previous job ID** and **next job ID**. This creates a doubly-linked list and so when a job is finished executing, it can add the next job in $O(1)$ time.

[1]Implementation provided by chashmap `https://crates.io/crates/chashmap`

3.3.2 Threads

Here is an overview of the important threads in `Prime Minister`:

Name	Description
Main Cluster Thread (MCT):	Heart of *Prime Minister* making all of the assignment operations and monitoring the overall state of the cluster
User Server:	A TCP server for users to send their requests to. Once a request is received, a (green) thread is created to handle the request.
Worker Server:	A TCP server for workers to send their requests to. Once a request is received, a (green) thread is created to handle the request.
Worker Client:	A thread pool for handling the MCT's worker requests.

Table 3.4: *Different threads in Prime Minister*

Green vs Kernel Threading

Green threads are "lightweight" threads scheduled by a runtime library (differing from the kernel threads, which are scheduled by the OS). Rust's standard library provides kernel threads, which are often a better option, however I/O operations spend a lot of their time blocked, waiting for data to be sent/received, and therefore research has shown green threads can achieve better performance for I/O [30]. Green threads introduce work-stealing schedulers, where the compute can be yielded between green threads upon waiting for an I/O operation allowing the underlying kernel threads to make progress, by executing a different thread. Also, in the pursuit of scalability, creating lots of short-lived kernel threads on-demand will introduce a lot of unnecessary overheads.

Prime Minister uses both types of threading:

Green Threads: threads created for each request received on the servers, threads created for every client request sent.

Kernel Threads: Main Cluster Thread, user server, worker server & worker clients.

Server Threads

The server threads expose a TCP server for both user and worker connections. Both servers follow the communication specification (§3.6). As stated above, to provide a more scalable, efficient system, green threads are spawned for each connection because:

- **The threads are I/O bound:** Majority of their time is spent waiting on I/O operations, so creating a kernel thread to simply block on several syscalls is a waste.

Instead, green threads use asynchronous I/O. Where an I/O call hands over control to the threading library, which reschedules the thread on operation completion. Meanwhile compute yielded to another thread.

- **Kernel threads introduce a lot of overhead:** Kernel threads are relatively expensive as they require the creation of a stack space, and require syscalls to allocate & deallocate. Green threads live in the heap and are a lot quicker to initialize than a kernel thread.

Worker Client Threads

```rust
pub enum WorkerUpdateType {
    Heartbeat,
    Cancellation,
    Submission(String) // Contains task ID assigned to worker.
}
pub struct WorkerUpdate {
    pub message: WorkerUpdateType,
    pub worker_id: String,
    pub ip_addr: String,
    pub ip_port: i32,
    pub retry_count: u32 // After 3 retries, message is reverted.
}
```

Listing 3.5: *Type definition of* `WorkerUpdate`*. These structs are placed on the message queue shared between the MCT and the client threads*

Instead of sending messages to workers on the main cluster thread (MCT), separate transmission threads are used, allowing them to asynchronously handle any responses, retries and any required cluster state updates. Sending on the MCT would both slow down the clusters' ability to react to new events and introduce scalability problems.
Communication between the worker threads and the MCT is brokered by a **message queue**: when the MCT needs to make a request to a worker it places this action (see Listing 3.5) onto the queue to be picked up by the transmission threads. A (kernel) thread blocks on the queue and creates a new (green) thread to be scheduled on a thread pool, which sends the request and processes any results.

Main Cluster Thread

The main cluster thread is the heart of *Prime Minister*, making all the assignment decisions and ensuring the robustness guarantees. Server & client threads in *Prime Minister* simply read and update the values in the relevant data structures, but do not make further options based on the received information. For example, upon task completion, the server thread only transfers the data into the master and marks the task as complete. Actions such as checking whether the enclosing job is complete are left for the MCT, so it can take further actions such as cancelling or adding jobs. This reduces the maximum response latency by minimising the number of (potentially contentious) locks that need to be held by any one request. Then the MCT can perform the actions that may require multiple locks to be held, when overall latency isn't as important.

These are the following actions that this thread takes:

- **Kick inactive users:** After 30 seconds of inactivity from the user (e.g. program crash), they are disconnected from the cluster and all of their data and workloads are removed.
- **Detect worker crashes:** Each workers' missed heartbeat count is updated by the client thread. Once the worker misses 7 heartbeats the worker is removed, any any running tasks re-queued.
- **Handle finished tasks:** Once a task is finished, it can be removed from the running tasks, and enclosing job's finish count is incremented. If the task is the last in the job to finish, then the next job is added to the job queue.
- **Handle halted tasks:** If a task has been halted, likely due to an uncaught exception raised in the closure, then the entire workload needs to be cancelled. The MCT cancels all of the tasks of the current job and any future jobs by adding cancel updates to the worker client queue. Finally, data for the workload can be removed from the cluster.
- **Create tasks from queued jobs:** After all tasks from the previous job have completed, the number of tasks developed for the job is known, therefore the cluster can generate the tasks for next job and place them on the queue.
- **Assign tasks to workers:** `Parliament` implements a FIFO scheduler, where it assigns tasks from the head of the task queue to any available workers, placing the submission request on the worker queue.
- **Send heartbeat requests:** Heartbeat requests are placed on the worker queue every 750ms. If a worker misses 6 heartbeats in a row (i.e. unavailable for >4.5s), then the worker is kicked and any tasks currently running that were running on it, are placed back in the tasks queue. These heartbeat requests ensure that the *Member of Parliament* module itself is reachable from the *Prime Minister*.

The current implementation is single-threaded, which prevents the possibility for a deadlock with another instance of itself. During evaluation, the MCT loop iterations typically took <1ms, so there was no immediate need to implement a multithreaded approach. Network operations are typically more likely to be a bottleneck in such a system, rather than lock contention.

Future work can be done to define a lock order which allows a multithreaded implementation, removing the potential bottleneck.

3.4 Parliament

The *Parliament* OCaml library is for the users of this framework to develop applications that run on the cluster. As stated in the conceptual model, developers are required to create a workload to submit to the cluster. To connect to the server, developers provide the user port and hostname of *Prime Minister* instance, via the command line.

3.5 Member Of Parliament

Member of Parliament is a Rust executable that communicates with both the *Prime Minister* and the user-created OCaml executables. Instead of a direct connection between the master and the OCaml executables, *Member of Parliament* exists to both monitor the executable in case of crashes and pre-empt a task in the case of an error. This separation works well when adding support for containers, as *Member of Parliament* can run directly on the host machines while communicating with the containers.

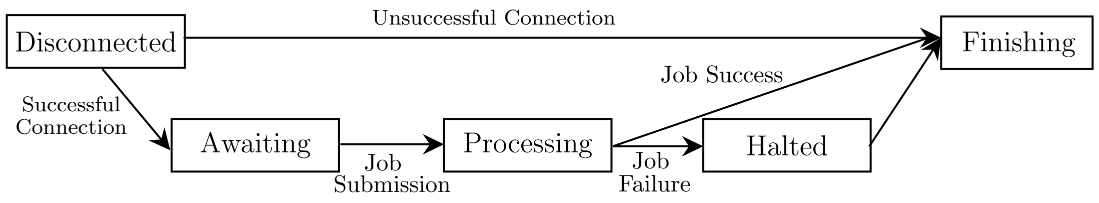


Figure 3.6: *State diagram of Member of Parliament*

As shown in Figure 3.6 *Member of Parliament* can be shown to be modelled as a state machine passing through states.

3.5.1 Communication with OCaml

For the OCaml process to know which closure to execute with data, the *Member of Parliament* modules communicate using a TCP socket with the running process. Executables run in a special **worker mode**, activated through environment variable. Once the OCaml process is running, it communicates with the server to both retrieve the data, pre-execution, and to send to the task status and data upon task completion, post-execution. After the process has sent its output, it is killed so its resources can be released back to the system. Communication is performed using protocol buffers (§3.6).

Future work could lead to the executable retrieving the data directly from Prime Minister to minimise network I/O, however this would limit efforts for data-locality optimisations.

3.6 Communication Design

Communication in any distributed system is very important. The WordCount benchmark, used in the evaluation (§4.1) averaged 9.3 messages sent per second with 4 workers. An inefficient method of communication, in both serialisation latency and payload size could reduce the efficiency and scalability of the system. Therefore, care must be taken to select the correct marshalling format that is both efficient for serialisation and compact for transmission. In many cases, I would be sending structured data between processes written in different languages. I, therefore, considered four marshalling options for unified communication:

1. Using OCaml's in-built marshalling library
2. Writing my own marshalling library both for Rust and OCaml
3. Using JSON (JavaScript Object Notation) [31]
4. Using protocol buffers (Protobuf) [32]

Option 1 only works between OCaml processes and requires a secondary solution for communication between OCaml and Rust processes. Option 2, if implemented correctly, should be the most efficient as knowledge about the standard format of the data to serialize it in an efficient way. However, it would add unnecessary extra work to the project and is not as extensible as other options for new data types.
Options 3 and 4 both involve using a data-interchange format for structuring the data. The main advantage is that they are both language independent with readily-available serialisation libraries for most languages. Protocol buffers [32] messages can be defined independently using *.proto files* and using provided tools, code can be autogenerated which allow you to serialize and deserialize the data. Comparison data on JSON and protocol buffers [33] show that protocol buffers generate both smaller binary files and offer shorter serialization/deserialization times when compared to JSON. I opted for option 4 due its performance characteristics without writing a custom solution. Both option 3 and 4 do not natively support data streaming, requiring all of the data to be available before serialization/deserialization, which can lead to an extra unnecessary memory usage.

3.6.1 Message Syntax & Semantics

Message sending in *Parliament* is performed using the request/response semantics. When sending bytes over a TCP socket, framing methods are required so the receiving party is aware of the end of a message. TCP provides no in-built capabilities for denoting the end of a message, this is instead handled by a higher-level protocol e.g. the length field in HTTP headers. To solve this problem, I use length prefixing: I prepend a big-endian 4-byte unsigned integer of the message length before sending the message. This provides a crude form of message corruption/error detection when the socket is closed before the message end, or the message cannot be decoded with the given length.

This leads to a maximum message length of 4GB per message, however, in big data processing libraries, large messages are avoided because of unreliable network and instead stored in distributed file systems, which offer their own consistency guarantees.

3.6.2 Messages

After designing the marshalling format, the next logical step is designing the data transfer objects to be exchanged by the different modules. Protocol buffer specification files (`.proto`) provide an easy language-agnostic method of specifying the communication. Appendix A shows the contents of the protocol buffer files which define the communication between the different modules of the system.

Unified Messages

Due to the number of different messages that a particular module could send to another, similar to length prefixing, I required a method in order for the receiver to determine which message it was being sent. Protocol buffers messages do not contain descriptions of their own types, therefore a method must be established to determine the message type before deserialisation can occur.

```
message SingleUserRequest {
    oneof request {
        CreateConnectionRequest create_connection_request = 1;
        ConnectionRequest connection_request = 2;
        JobSubmission job_submission = 3;
        DataRetrievalRequest data_retrieval_request = 4;
        JobStatusRequest job_status_request = 5;
    }
}
```

Listing 3.7: *Example of a unified message.* ***`SingleUserRequest`*** *is sent from the Parliament library to the master*

A message type identifier could have also been prepended to the message with the length, however this would require maintaining a map of this information across four codebases, which can easily lead to accidental alterations and difficult to detect errors. Instead, I added another message definition for each set of directional unified messages (see Listing 3.7) which contain at most one of the individual messages. These provide a language agonistic method of specifying the type, with the deserialisation provided for free, by the protocol buffer library.

⁂

The following sections explain the implementation of the extensions of `Parliament`, both implemented to be independent, optional add-ons.

3.7 Job Isolation

Resource isolation is an important goal in cluster computing frameworks [18]. The initial implementation of the library however offers no isolation, user executables run directly on the OS allowing them to, intentionally or not, affect other running jobs on the same machine and more. Containerisation offers the ability to create isolation units easily, providing a significant amount of protection and preventing accidental interactions between tasks.

3.7.1 Limitations

However, containerisation is not a free lunch, it introduces a performance hit [22], as well as some qualitative drawbacks:

1. **Longer compilation time:** Before being able to run your application on the cluster, a container image needs to be created, and likely pushed to a central registry which takes a significantly longer time than compilation. On the evaluation machine (see Table 4.2), the `Parliament` WordCount benchmark took 2.39s to compile, while the corresponding container took 49.85s.[2]

2. **Extra system requirements:** The user's host machine (and all of machines in the cluster) running the application, require the Docker runtime installed on their machine, to submit containerised jobs. This is due to limitations found in OCaml's Marshal library requiring the exact executable to transmit closures.

3.7.2 Benefits

Apart from the obvious benefits of isolation from other running jobs, the main other benefit is standardisation. In a cluster where heterogeneity is common, containers provide an easy solution of providing consistent environment for all tasks. This can be useful when the tasks require external dependencies.

3.7.3 Implementation

The implementation is analogous to standard `Parliament` jobs, instead of starting a new process, the *Member of Parliament* modules connect to the Docker daemon and start a

[2]Building containers is a mostly single-threaded process, hindering its performance on the evaluation machine which only boasts a clock speed of 1.9GHz. However, their portability allows them to be built on a faster machine then run anywhere.

new container. The process for retrieving and sending data is exactly the same (protocol buffers over a TCP socket), and the containers are removed on completion.

3.8 State Machine Replication

Prime Minister presents a single point of failure; a process crash or network connectivity issue could lead to a significant progress lost. To provide master replication, I adapted an approach introduced by Paxos' Parliament protocol [34]. Paxos is a method of achieving agreement in replicated state machines by achieving progress through majority voting. In preparation for this, each object in `Parliament` (task, job, user, worker) was modelled as a state machine (see Listing 3.8) enabling me to model the *Prime Minister* as a state machine. Each cluster state is a vector of the states of the units in the cluster:

$$< \{w|w \in Users\}, \{x|x \in Workers\}, \{y|y \in Jobs\}, \{z|z \in Tasks\} >$$

```
pub enum TaskStatus {
    Awaiting, // Awaiting for a task assignment from cluster
    Running(String), // Running a task stated in string
    Completed, // Completed task, about to restart
    Halted, // Task has resulted in an error
    Cancelled // Task has been cancelled before completion by cluster
}
```

Listing 3.8: *Excerpts of code from* `model.rs` *showing the different states of a task*

Additionally, to avoid problems such as duplicate task assignment, both *Prime Minister* and *Member of Parliament* offer *at-most-once* message delivery semantics. This is modelled by a self-loop for duplicate messages (see Figure 3.9).

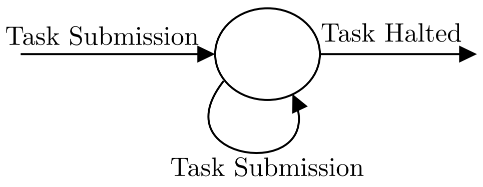


Figure 3.9: *Example of at-most-once message delivery semantics implemented as a state machine*

Replicated State Machine's [35] over-arching idea is: if each replica of *Prime Minister* start in the exact same state, and process requests/inputs in the exact same order, then they should pass through the same states and hence produce the same output. This is called **Replica Coordination**. From the paper, it is shown we can decompose this problem down into two separate parts:

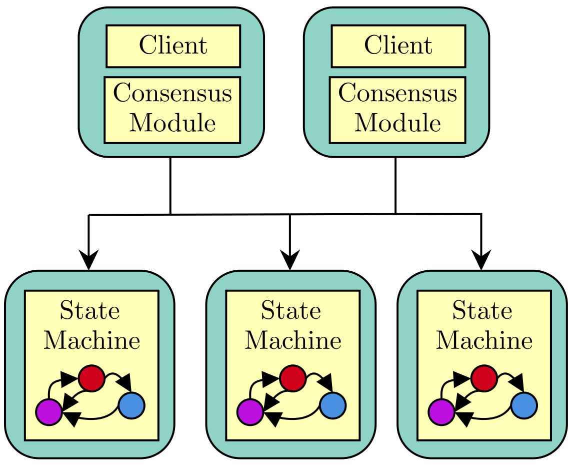
1. **Consensus:** Every non-faulty master receives every request
2. **Order:** Every non-faulty master processes requests in the same total order

3.8.1 Consensus

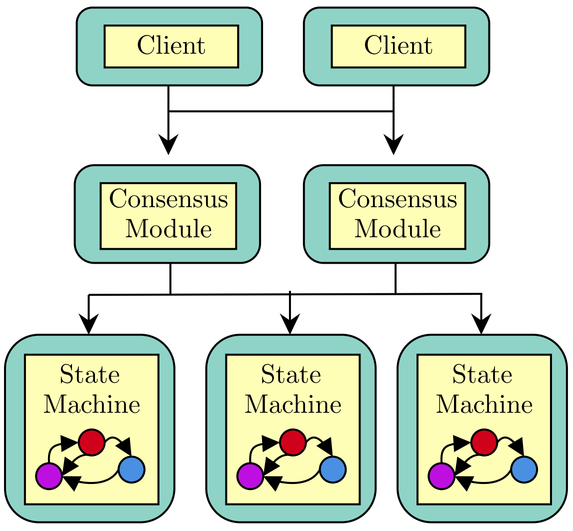
Here 'client' is used to represent the users and workers, i.e. from the Parliament library and Member of Parliament modules. While 'replica' is used to represent the Prime Minister module.

To send messages to the masters to achieve consensus, there are two main system architectures to choose from:

1. Clients talk directly to the master replicas (Figure 3.10a)
2. Clients talk to a consensus module which talk to the master replicas on behalf of the clients (Figure 3.10b)



(a) *Option 1: clients talking directly to the master replicas*



(b) *Option 2: clients talking to a consensus module which in turn talks to the master replicas*

Figure 3.10: *Diagrams of Consensus options*

Option 1 is a simpler solution due to its similarity to the current system layout, however, it requires that all clients know about all replicas, requiring some form of discovery. Also, the mandatory ordering between clients' messages may require them to talk directly to each other.
Option 2 moves the responsibility of managing the masters away from the client which removes the need for the clients to even be aware of the replication and the need for initial code changes to the OCaml *Parliament* library and *Member of Parliament*. This does, however require the *Consensus* modules to be replicated also, to remove any single point of failure.

I opted for option 2 due to the detached consensus responsibility and it removed the potential need for users to talk to workers for ordering. Also, as the *Parliament* library is written in OCaml, it would require requests to be sent to each replica sequentially, drastically hurting performance.

3.8.2 Ordering

Paxos requires each message (i.e. request) proposed to have a unique and maximal – greater than any previous – number. However, this is based on several assumptions which can be relaxed:

1. **Every message sent may potentially conflict with every other message:** Messages (in `Parliament`) will only conflict with each other if they access the same data, i.e. messages from different users cannot conflict with each other and messages from different workers cannot conflict with each other.

2. **There is an equal chance of any message being sent at any one time:** Certain messages can only be sent at particular stages in a connection lifecycle, therefore, cannot conflict with certain other messages. E.g. *CreateConnectionRequest* can only be sent when a user is connecting to the server.

3. **Senders can have over 1 open connection to the master at any one time:** The users and worker processes send messages in a single-threaded fashion, therefore a user's message cannot conflict with itself, similar with workers.

These assumptions lead to the conflicting operations table:

User → Master	
CreateConnectionRequest	None
ConnectionRequest	None
JobSubmission	None
DataRetrievalRequest	WorkerFinishedRequest
JobStatusRequest	WorkerFinishedRequest

Worker → Master	
WorkerConnectionRequest	None
WorkerFinishedRequest	DataRetrievalRequest JobStatusRequest

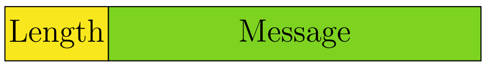
Table 3.11: *List of messages and their respective conflicting messages*

For the messages that can feasibly conflict with others, the *Consensus* module will use a *message ID* which enforces a total order for operations. Each *Prime Minister* module will maintain their own message ID counter as a Lamport's [36] clock: on encountering a message with a larger ID, it will accept and update its internal counter (to the received value); else, reject the message.

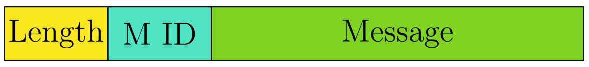
Two-phase commit

The original Paxos paper specifies a two-phase commit, in which a change is proposed (to all replicas) and then committed. This would require a large change to the semantics of *Prime Minister*. Instead, *Consensus* uses the majority vote of the *Prime Minister* modules (accept or reject) as the new state of the system.

3.8.3 Changes to Communication Design



(a) *Message payload from clients to masters/consensus modules*



(b) *Message payload from consensus modules to masters*

To send the *message ID* from the *Consensus* module to the *Prime Minister* module, the communication format was adapted. Besides to sending the message length, the *Consensus* module adds a *message ID*, a big-endian 4-byte unsigned integer to the payload. Figure 3.12a shows the previous design, employed by the workers and users and Figure 3.12b shows the adapted design used by the *Consensus* modules. This message ID field is added to all messages, and set to 0 for operations that do not conflict with any others.

3.8.4 Randomness

To implement replicated state machines correctly, randomness must be removed from the state machines. In the current initial implementation, the *Prime Minister* module assigns a random ID to users and workers. Assigning a different ID to the same user/worker will lead to lost results and users unable to access their data.

To fix this, a hidden field is added to the **CreateConnectionRequest** and **WorkerConnectionRequest** allowing an ID override (see Appendix A). If the *Prime Minister* modules receive a request with the ID override field set, they will use the value as the ID for that worker/user. This allows the *Consensus* modules' upon receiving either connection request to generate an ID for all replicas to use. These can be guaranteed to be unique by the module maintaining a list of used IDs.

3.8.5 Bi-directional Communication

Communication between the *Prime Minister* and *Member of Parliament* modules is bi-directional. In the current architecture, this would lead to all of the masters sending (duplicated) requests to all of the workers, leading to unnecessary communication and the replicas after the initial potentially receiving unexpected responses.
To solve this problem, we elect a *leader replica* for sending the task requests to the workers, the remaining replicas are named *follower replicas.* The ability to send **WorkerTaskSubmissionRequest**s and **WorkerTaskCancellationRequest**s is now solely given to leader replicas, while any replica can send a **WorkerHeartbeatRequest**. This leader is elected and monitored by the *Consensus* modules and run the re-election procedure to enforce the invariant that there is always exactly one leader. The follower replicas gather information about which tasks are running on each worker, by the information returned by the **WorkerHeartbeatResponse**.
The masters are now required to contain state about whether they are running in consensus mode and whether they are a leader/follower replica. This also required the introduction of dedicated communication between the *Consensus* and *Prime Minister* modules. A new message: **ConsensusRequest**, (see Appendix A.4) was added in order for the communication specification, which allows the *Consensus* modules to instruct the masters whether they are the leader or not.

3.8.6 Eventual Consistency

By adding a leader replica, the (strong) consistency guarantees have to be relaxed, since the follower replicas are not consistent at all times. The follower replicas are now only aware of task assignments after sending heartbeat requests.
Eventual consistency [37] is a form of weak consistency where the system guarantees that if no new updates are made to the object, *eventually* all accesses will return the last updated value. *Consensus* exhibits this behaviour as the follower replicas are initially unaware of task assignments. However, the replicas will (periodically) send a heartbeat request to the workers and receive information about the task assignment, *eventually* reaching the same state as the leader. This is a slight design change from typical eventually consistent systems, where the writes are propagated directly to the other replicas. However, any form of gossiping protocol between replicas would break replicated state machine semantics.

Proving unchanged semantics

While a formal proof that eventual consistency provides that same semantics is required to fully verify the system, I only provide an informal proof due to the scope of the project:

Eventual Consistency Guarantee. The eventually consistent system does not need to provide equivalent semantics to a strongly consistent one, it is simply required to either **reach the state of strongly consistent system** or **evidently be in an inconsistent state** (to the *Consensus* modules).

A follower replica is left in an inconsistent state when a worker has passed through two states before the replica sends a heartbeat request, i.e. the intermediate state is *lost.*
The only possible state transitions would be:

Case 1: `Awaiting` → `Processing` → `Halted`
Once a process reaches the `Halted` stage, it sends a **WorkerFinishedRequest** to the replicas.
If we **have** received that request, then we can determine that the process has successfully executed the task with the ID, and we reach the same state as any strongly consistent system. No information was lost.
If we **haven't** received the request, then we are in an inconsistent state, and since we didn't receive the request, we are about to be removed from the pool of replicas from the *Consensus* module, as we are unreachable from *Consensus* module. The overall system progresses in a consistent state.

Case 2: `Awaiting` → `Processing` → `Cancelled`
In order for a process to be in the `Cancelled` stage, another worker executing a task from the same job has halted and the leader replica has cancelled the workload.
If we **have** received the request from other worker, then we too have cancelled the job, leaving us in the same state as the leader replica. No information was lost.
If we **haven't** received the request from the other worker, i.e. the job is not cancelled then we in an inconsistent state, and will send back a response with $'request\_processed' = false$. Thereby being in a visibly inconsistent state.

Case 3: `Awaiting` → `Processing` → `Finishing`
An analogous reasoning follows as explained in case 1.

From this we can deduce that the consistency guarantees have not weakened. □

Limitations

Since writes are not synchronously propagated between replicas, there is a possibility of data loss. In this case, this would lead to task assignments being lost, which means computation time being wasted. Heartbeat requests are sent every 750ms, which allows the maximum computation time loss to be easily quantified:

$$\text{Max computation time loss (s)} = 0.75 * \#\text{No of workers}$$

3.8.7 Consensus Replication

To finally achieve no single point of failure, the *Consensus* module needs to be replicated itself. This is accomplished by a simple gossiping protocol between the modules. In this protocol, they share information about what masters are still consistent and which *Consensus* modules are still active.

```
message HeartbeatRequest {
    message Consensus {
        int32 id = 1;
        string ip = 2;
        int32 port = 3;
    }
    repeated Consensus consensuses = 1;
    message Master {
        int32 id = 1;
        bool master = 2;
        string ip = 3;
        int32 worker_port = 4;
        int32 user_port = 5;
        bool active = 6;
    }
    repeated Master masters = 2;
    Consensus me = 3;
}
```

Listing 3.13: *Specification of HeartbeatRequest sent between consensus modules*

Every 2000ms, or on state change (see Listing 3.13) each *Consensus* module shares its current state of the world with every other known *Consensus* module. On receiving this message, each module will take an intersection of their own view of the world with the received view, ensuring that the receiving module is retained and send it back.
Commercial tools such as `Kubernetes` [38] and `etcd` [39] exist as commercial, open source solutions for adding container orchestration and replicated state. However both are substantial systems for a wide range of use-cases, however `Parliament` is focused on cluster computing, and it didn't make sense to use these large systems for the small subset of features they provide.

⁂

One of the main advantages of `Parliament` is the separation between the cluster scheduler and the OCaml library. This presents potential for equivalent *Parliament* implementations, in different languages, which follow the conceptual model and communication design presented in this chapter.

3.9 Repository Overview

The high-level structure of the project has already been discussed (§3.2), so I will briefly give an overview of the code structure here.

- `Protobuf Specification Files` **Protocol Buffers Files**
- `Parliament` **OCaml**
  - `lib`: *Parliament* library for OCaml written by myself
    - `Core`: Code generated from Protobuf using ocaml-protoc [40] tool
  - `test`: Unit tests for *Parliament* library
- `Member Of Parliament` **Rust**
  - `src`: *Member of Parliament* module written by myself
    - `protos`: Code generated from Protobuf using rust-protobuf [41] library
  - `tests`: Integration tests for *Member of Parliament*. Unit tests are found inside source files as per Rust code standards.
- `Prime Minister` **Rust**
  - `consensus`: *Consensus* module written by myself
  - `minister`: *Prime Minister* module written by myself
  - `shared`: Shared libraries and code used by both *Consensus* and *Prime Minister*
- `Integration Tests`: **Bash & Python** Tests that involving running sample workloads on the different modules.
- `Evaluation Code`: **OCaml** OCaml code used in evaluation tests.

Chapter 4

Evaluation

In this chapter, I will analyse my system, comparing its performance to both standard OCaml and established frameworks, achieving my success criteria: it is possible to reduce the running time of a MapReduce program written in OCaml. Moreover, I will justify my methodology, taking cues from proven evaluation of similar systems. I will analyse and explain the parts of the implementation leading to these results, allowing me to establish the strengths, weaknesses and opportunities for future work. My results analysis shows that `Parliament` can successfully use multiple cores to speedup execution, decreasing the latency of many OCaml applications.

4.1 Evaluation Method

I used the tests outlined in Intel's HiBench Benchmark Suite, a standard benchmarking suite for big data processing frameworks [42]. The suite is used to test the capabilities of systems based around the MapReduce model. `Parliament` is not built only for MapReduce jobs, but rather a wider class of distributed execution; however, MapReduce jobs are most widely understood in industry.
I ran three tests, taken from the suite's micro benchmarks. These programs represent a large subset of real-world MapReduce jobs. I tested the performance and overheads of such systems:

- **WordCount**: Used in the original MapReduce paper [16], it extracts a small amount of interesting data from a large dataset, consequently it is mostly CPU bound. To ensure relevant and verifiable results, I used three files, each containing an eBook [1] concatenated several times, of file sizes 200MB, 400MB & 600MB. Larger files would eventually be limited by the slow I/O speeds of the evaluation machine. They would result in an execution where the I/O operations, which are typically single-threaded, are responsible for majority of the execution, limiting any parallelisation efforts. `Parliament` implementations partitioned each file into 20MB sets.

[1]The Project Gutenberg EBook of Knights of Art, by Amy Steedman

- **Sort**: Also used in the MapReduce paper, it transforms a set of data into a different form. Since the job input and output are of equal size, it is mostly I/O bound, with a moderate CPU utilisation. Sorting involves taking 5 files, each 50MB containing a list of unsorted words and producing a file with the same words sorted. The computation is transforming data into a different form of equal size, meaning it will be more I/O intensive than WordCount. The `Parliament` implementations partitioned each file using the first character as a rudimentarily hash function.
- **Sleep**: This benchmark creates multiple tasks run the sleep function. I used this test to compare the performance of the scheduler and calculate the overhead introduced by the cluster. Faster task assignments lead to a lower overall latency of workloads, which benefit all jobs. The test created 60 tasks each to sleep for one second. All other time costs (submission & initialisation time) were considered negligible.

For each experiment, I compared 5 different executions (see Table 4.1).

Name	Description
Standard OCaml	A standard OCaml program.
Parliament	Parliament running with a single-master and tasks running directly on OS.
Parliament w/ Docker	Parliament running with a single-master and tasks running in Docker containers. Time to pull images was not included in experiments.
Parliament w/ Consensus	Parliament running with three masters and three consensus modules and tasks running directly on OS.
Apache Spark [8]	Spark running in Standalone mode [43] with individual executors each with a single core. Code written based on examples provided by HiBench repository [44] (see Appendix B).

Table 4.1: *The different system setups used in the evaluation experiments*[2]

Each experiment was performed 100 times to accurately measure the variance to demonstrate the reliability of the systems. For each test, a sequential algorithm was written, and then a separate parallel variants created each library's primitives. The different `Parliament` variants allowed me to accurately calculate the overheads for each extension. Spark [8] is an existing cluster computing library, built for performance, created for MapReduce and similar workloads. It works by providing a high-level abstraction called Resilient Distributed Datasets. Comparing raw performance between `Parliament` and Spark would be pointless due to the misalignment of priorities between systems. It does, however, provide a standard for a similar system, useful for comparing characteristics.
All experiments were run on a single machine (see Table 4.2) to reduce any externalities introduced by networking. Separate qualitative tests were used to ensure `Parliament` operates correctly across multiple machines.

[2]Three instances are used to achieve quorum, to maximise the chance of a majority in case of a split.
[1]Calculated using the `dd` command, copying to and from `/dev/null`
[2]Ubuntu used as it provided asynchronous I/O operations in kernel.

CPU & RAM	
CPUs	4x 12 Core 1.9Ghz AMD Opteron™6168
Cache	4x L1 1.5 MB
	4x L2 - 12 x 512 KB
	4x L3 - 12 MB
RAM	64GiB
I/O	
Write Speeds	570MB/s
Read Speeds	76.7 MB/s [1]
Operating System	
Version	Ubuntu 18.04.1 LTS[2]
Load	Minimal

Table 4.2: *Evaluation Configuration*

4.1.1 Reproducibility

Page Cache

The Linux kernel implements a page cache, to minimise the disk I/O [45, Chapter 15]. Upon being placed in memory, files remain there until eviction by a running process or further page caching. Two benchmarks include reading & writing from a file, therefore to guarantee experiments were reading from disk, the cache was flushed between iterations.

File Placement

File placement on hard disk drives is important as it can influence disk speeds. Hard disks are split up into tracks, however the longer outer tracks contain more sectors than the shorter inner tracks. As the time per rotation is constant, more sectors can be read on the outer tracks, leading to a significant speedup in disk speeds [46]. To ensure a similar read speed, the files were read from the same location for all experiments.

4.2 Analysis of Timing Results

Confidence bars for timing results (see Figure 4.3, 4.4 & 4.5) were too small to be identified on a plot, so (sample) variance is provided in Table 4.6. Spark Sort results omitted as unable to identify bug in Spark code, leading to poor performance.

	Standard	**Parliament**	**Parliament w/ Consensus**	**Parliament w/ Docker**	**Spark**
WordCount	11.5	3.38	3.45	47.3	4.82
Sort	6.60	0.372	0.513	153	N/A
Sleep	N/A	0.00000625	0.000171	N/A	0.00452

Table 4.6: *Variance of execution experiments*

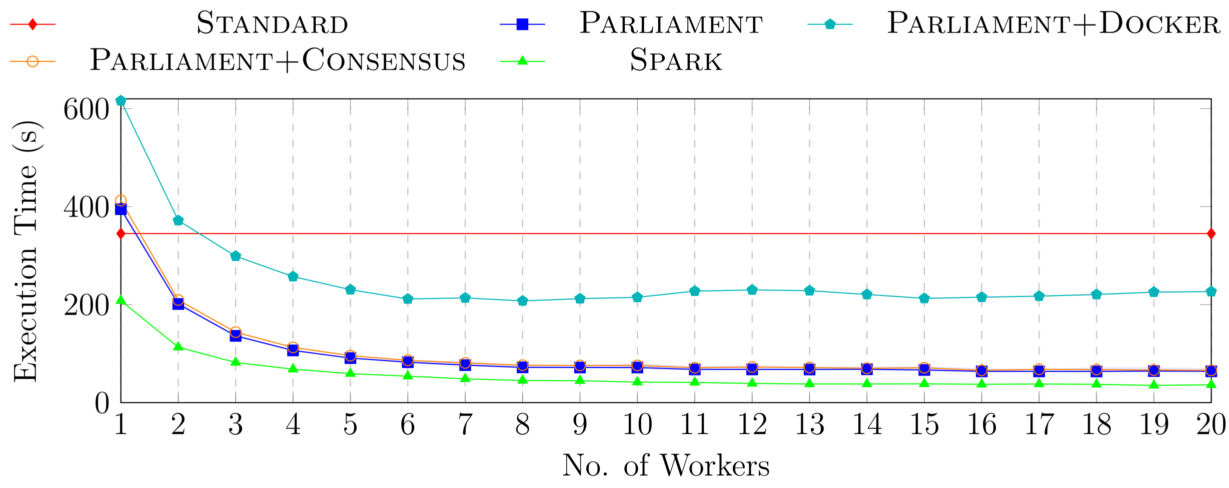


Figure 4.3: ***WordCount:*** *Results of execution time of the fixed-size WordCount benchmark between standard OCaml execution and all of the variants of* **`Parliament`** *& Spark*

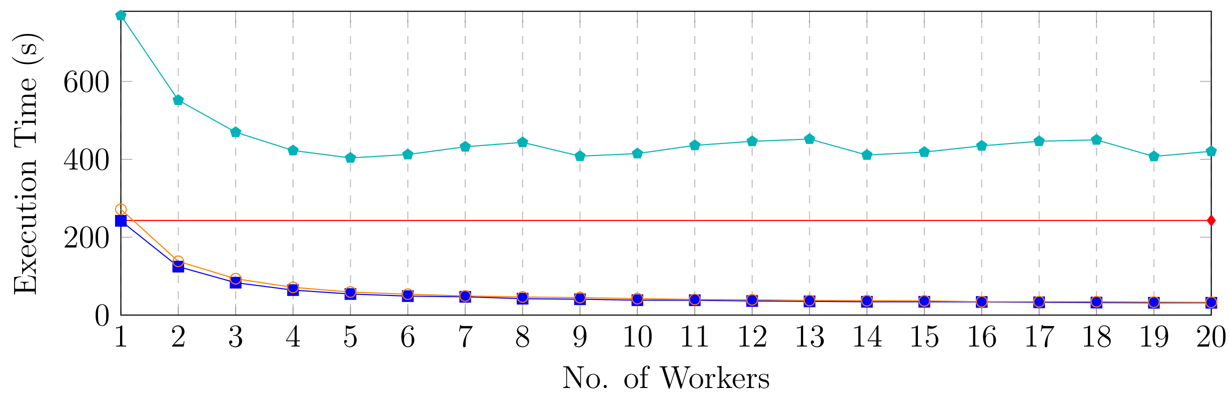


Figure 4.4: ***Sort:*** *Results of execution time of fixed-size Sort benchmark between standard OCaml execution and all of the variants of* **`Parliament`** *& Spark*

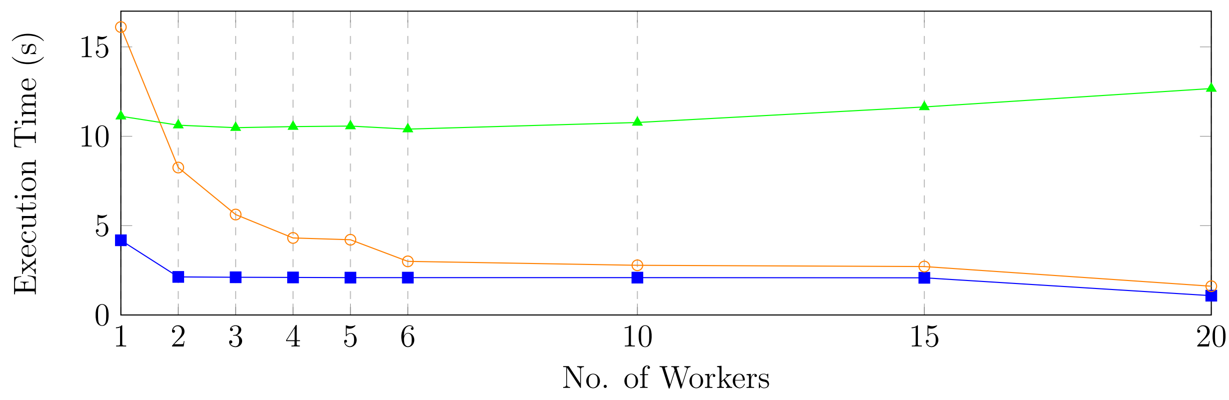


Figure 4.5: ***Sleep:*** *Results of execution time delta of sleep benchmark. Time is calculated by taking difference actual execution time and 60/no. of nodes.* [3]

4.2.1 Parliament

Both the WordCount and Sort tests (see Figure 4.3 & 4.4) show a significant speedup between `Parliament` and the standard OCaml execution. `Parliament` running with just two workers results in a 42% speedup, a demonstration of my success criteria being achieved. Sort's curve showed a surprisingly ideal behaviour for the gains that can be made from parallelism. This can attributed to the amount of parallelism extracted from Sort: both from the number of files and the number of tasks created, increasing the occupancy level of the cluster. Also, communication between the Sort tasks is comparatively smaller, only sending an average of 2 MB to each map task compared to the 20MB payload of the WordCount map tasks.

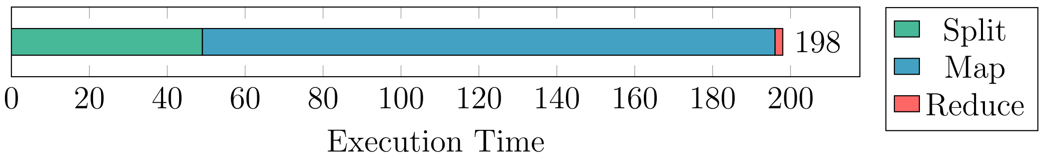


Figure 4.7: *The share of execution time between the 3 stages in Parliament. Results are for a 600MB WordCount workload running in a 1 worker cluster*

As expected, the large speedup gains when increasing from 1 to 2 workers cannot be expected when increasing from 15 to 16 workers. This behaviour is exhibited across the board. This is explained by Amdahl's law [15] which gives the theoretical execution time limit of a fixed task that can be expected if executed in parallel. The `Parliament` WordCount is not a wholly parallelised execution, and is split into 3 stages: split, map & reduce (see Figure 4.7). 26% of this execution is sequential, reading/splitting and writing/reducing a file), giving a speedup limit of 74% per file.

`Parliament` also reduced the variance in latency in both single and multiple master clusters, providing surprisingly more reliable results than standard OCaml. The use of Rust meant that `Parliament` cluster memory management performed similarly whether with 1 or 20 workers.

4.2.2 Docker

The Docker results were not as promising as initially expected. After further analysis, the main bottleneck was identified as the container start-up latency, which was an average of 3942ms. [1] [2] This result is exemplified in Sort, which creates 140 tasks (compared to WordCount's 69), where the startup costs completely dominate the total execution time, suppressing any performance gains obtained by parallelising the execution.
Upon further investigation, the main reasons established for this overhead were:

[1] Calculated by when running by *Member Of Parliament* a WordCount job with 1 worker
[2] Time includes green thread setup and destruction time

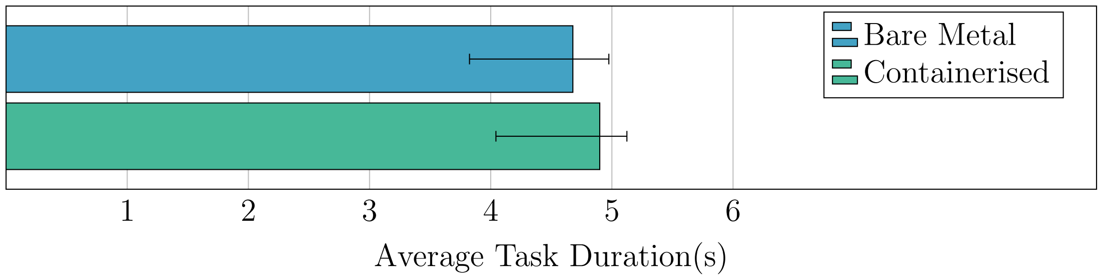


Figure 4.8: *Comparison of WordCount's map tasks latency [4], excluding startup time*

- **Slow Read Speeds:** Containers are significantly larger (file sizes) than their executable counterparts. The evaluation machine has comparatively slow read speeds, which bottlenecks the Docker daemon when it loads the image into memory. A similar test on a machine with faster disks[5]produced start-up times of ~900ms.
- **Rust Docker Library:** The Rust Docker client forces the developer to use a green threading library, which comes with its own initialisation start-up cost when the native backing threads are created, which further increases the start-up time cost.

This overhead only existed at startup, and once running, the Docker jobs were an average of 4.5% slower than the bare-metal executions, (see Figure 4.8) producing results much more aligned with the expected results from research [22].
The performance gap experienced between the paper's performance and actual empirical performance can be attributed to Docker networking. Docker provides virtualised networking which introduces unnecessary copying of packets, decreasing the network performance.
This unpredictability of the start-up cost also led to a large increase of the variance in the timings of the `Parliament+Docker` results. As above, this reduced on a machine with faster disk read speeds.
The startup cost impact can be reduced by creating fewer tasks. In its current form, the WordCount benchmark creates 69 tasks. but by doubling the input size for map tasks, decreasing the task count to 39, we can achieve better results. Achieving an average of 561s execution time compared to the above's 616s, with 1 worker (a 9% decrease).
This result did not carry to bare-metal executions: splitting into 40MB blocks incurred an additional 4s cost for a singular worker.

4.2.3 Consensus

The `Parliament+Consensus` results (see Figures 4.3, 4.4 and 4.5) provide reassuring results about the viability of adding multiple masters support to the `Parliament` system. It added a minimal constant overhead, which was expected, with >4x increase in communication between components in the system, and an increase in the variance of each

[5]Evaluation performed on my personal laptop, with a 2.6GHz dual-core CPU and read speeds of around 650MB/s

[5]Measured using time duration between TCP requests to *Member Of Parliament*

benchmark. This is due to the number of moving parts, increased computation and non-determinism introduced by running a multiple-masters system.

4.2.4 Sleep

Both Spark and `Parliament` [6] implement a work-conserving FIFO (first-in-first-out) scheduler and therefore should have comparable performances when scheduling. However, the `Parliament` cluster (see Figure 4.5) allocates jobs much faster than the Spark. Initial analysis suggests the `Parliament` scheduler is much more performant than the Spark's. However, the small change in Spark's performance when comparing 1 worker to 20 suggests there is an important setup cost introduced by Spark, which is non-existent with `Parliament`. For workloads that do not communicate a significant amount of data, `Parliament` adds a minimal overhead to the execution (∼1s for 20 workers).

4.3 Analysis of Fault-Tolerance

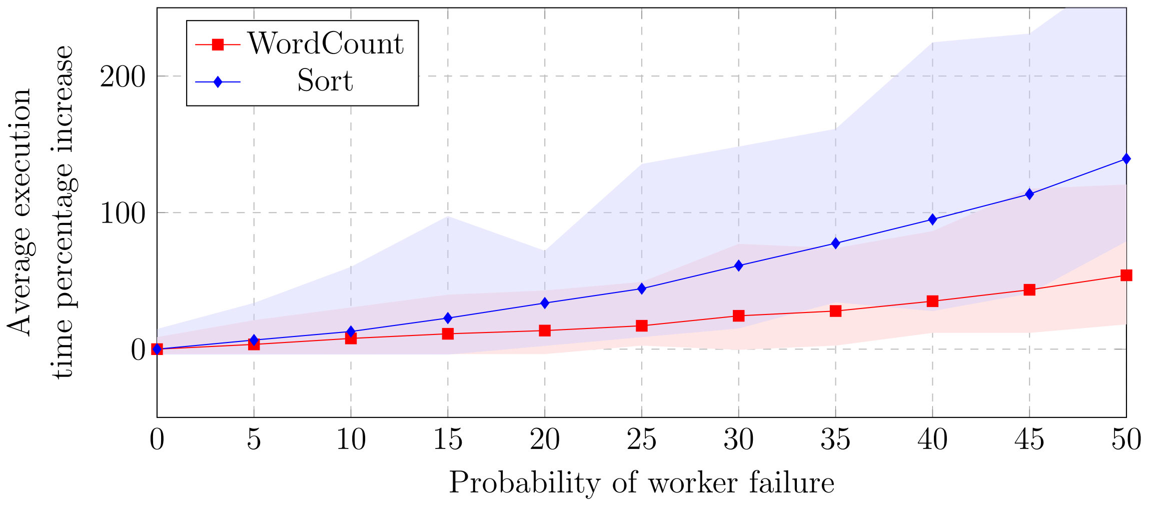


Figure 4.9: *Average execution time increase due to worker failures at a given probability.*[7] *Confidence intervals given as shaded regions.*

In addition to qualitatively ensuring the system provides (worker) fault-tolerance, it is important to ensure faults do not cause a significant impact on the execution time. To simulate failure, processes were killed on task assignment according to the failure probability, leading to a heartbeat timeout.

The results (see Figure 4.9) show comparatively linear results for the WordCount. Sort, however, creates significantly more (short-lived) jobs, and suffers because majority of its tasks perform little progress, executing for less than 400ms (an order of 10 shorter than WordCount). Leading to a success/failure cost ratio which is comparatively large compared to WordCount's, explaining the larger overhead gradient.

[6] Docker results ommitted due to large startup cost.

[7] Sleep omitted as it is not a typical distributed workload, so results aren't informative.

4.4 Analysis of System Utilisation

Finally, it is important to carefully look at the system resources' overhead that is introduced by using distributed computation. Efficiency is an important goal in cluster computing frameworks [18], and it is therefore useful to quantify any overheads introduced. Figure 4.11 shows the CPU[8]and memory usage of the different runs under the WordCount benchmark. Table 4.10 also shows the system utilisation time products.

The benefits of parallelism are realised here where WordCount is limited to a single core in the standard OCaml execution (2.04% of the total machine capacity). However, `Parliament` and Spark are able to use more resources, enabling them to complete execution in significantly less time. As expected, due to lack of communication required, the Standard OCaml execution uses less total CPU time. However, as `Parliament` kills processes, instantly returning memory back to the OS, it yields a more efficient use of memory. `Parliament` uses 44% less memory, executing in 31% less time, with only a 12% increase in CPU usage. While `Parliament+Consensus` results provide an acceptable overhead, `Parliament+Docker` results are affected due to the additional network and file I/O involved with containers. Again we can observe the effects of the containers' slow startup time, where the CPU and memory usage reach periodic peaks as the workers start their tasks almost in phase with each other.

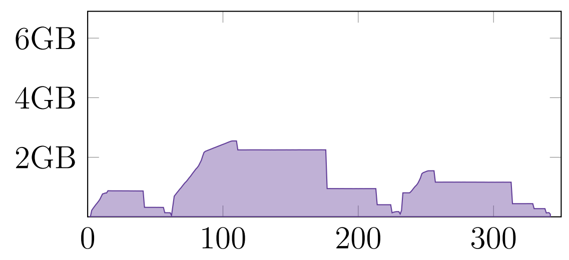
	Memory Time Product (GB s)	**CPU Time Product (Core s)**
Standard OCaml	493	337
Parliament	271	378
Parliament w/ Consensus	485	389
Parliament w/ Docker	602	966
Spark	363	467

Table 4.10: *CPU and memory time product of each execution.*

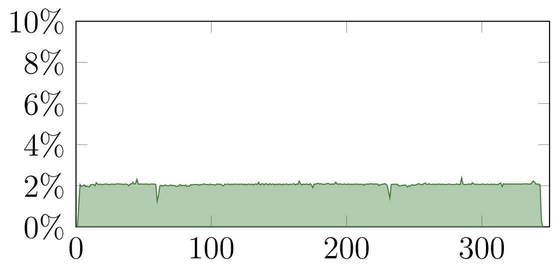
4.5 Summary of Findings

I have shown that it is possible for an OCaml execution to achieve performance gains by distributing it over multiple cores. Furthermore, `Parliament` is the first functional distributed library to both provide a functional programming model for multiple languages and supporting a shared processing cluster. `Parliament` successfully decreases the latency of workloads, achieving more than 7.5× and 5× speedup for Sort and WordCount respectively, while using significantly less memory and a minimal increase in CPU usage. This also opens the door up to variety of applications and use cases, which before did not consider OCaml as a natural candidate.
My extensions provided the desired assurances with the expected overhead and any drawbacks identified have been throughly explained and a solution provided and evaluated.

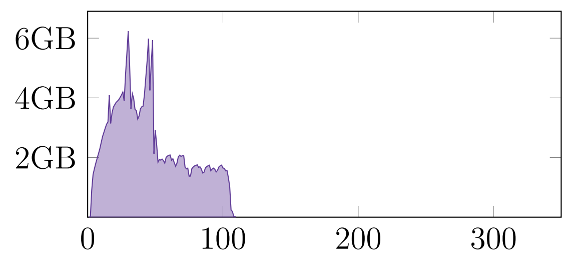
[1]CPU is given a percentage of the maximum throughput of all 48 cores



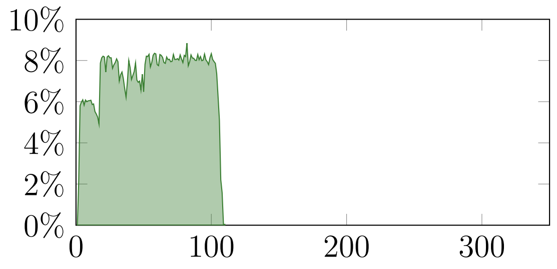
(a) *Standard OCaml: System memory usage*



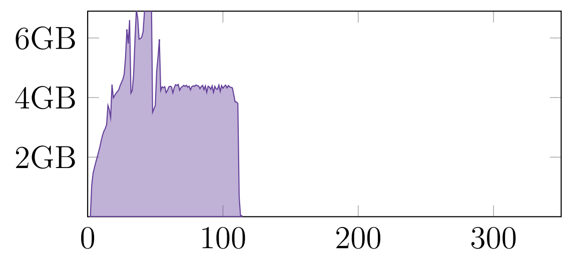
(b) *Standard OCaml: System CPU load*



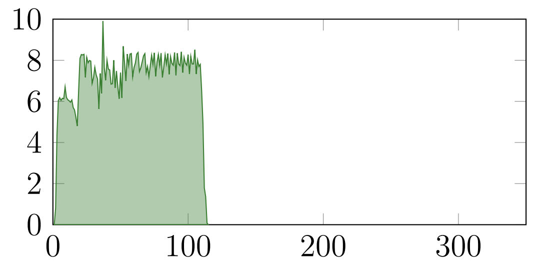
(c) *Parliament: System memory usage*



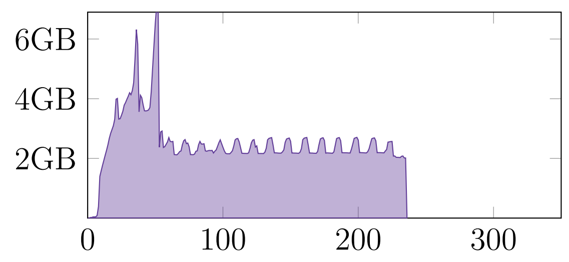
(d) *Parliament: System CPU load*



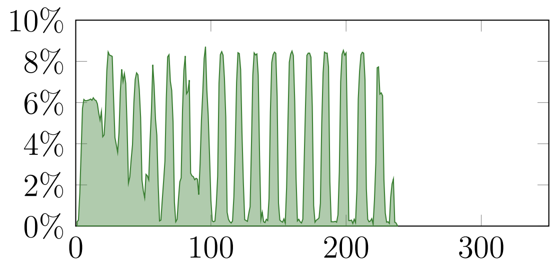
(e) *Consensus: System memory usage*



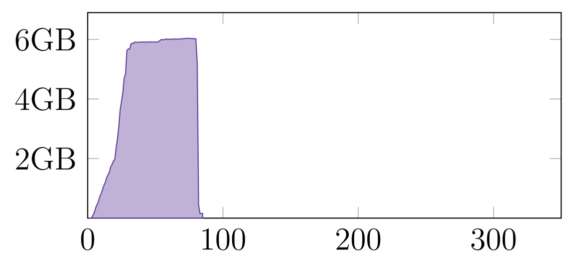
(f) *Consensus: System CPU usage*



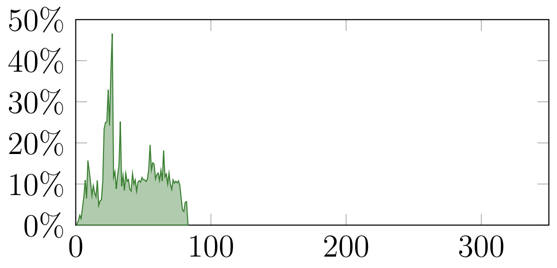
(g) *Docker: System memory usage*



(h) *Docker: System CPU load*



(i) *Spark: System memory usage*



(j) *Spark: System CPU load*

Figure 4.11: *System memory utilisation during WordCount benchmark. Parallel systems running with 4 workers. Measurements given as difference above baseline utilisation. CPU given as % of total capacity.*

Chapter 5

Conclusion

The results presented in this dissertation have fulfilled the hypothesis stated at the start of the project: shown by extensive tests, `Parliament` can decrease the latency of an OCaml program, allowing parallel execution across multiple cores, with minimal overhead. I can comfortably state to have met the success criteria laid out in the project proposal and the system requirements established in chapters 1 and 2.

5.1 Summary of Results

Extensive empirical evidence show that `Parliament` successfully takes advantage of multiple cores, decreasing latency of several typical parallel workloads. It also successfully handles failure crashes of any component in the system and isolates running jobs from each other. The implementation is empirically shown to achieve a running time improvement greater than 5-fold when running on a multicore machine and boast up to 6x faster than the Spark's default scheduler.

My extensions, job isolation and multiple masters support, were successfully implemented and worked interchangeably with the original system. The job isolation results were unfortunately limited by the evaluation machine. However I have sufficiently demonstrated that without the disk bottleneck, the Docker results are comparable to bare-metal. The multiple masters results were also very promising, the system proved it was able to recover from faults in less than one second.

5.2 Significance of Results

This system proves the viability of OCaml as a language of choice for big data processing. Developers can take advantage of the benefits provided by OCaml to build distributed jobs backed by a strong type system. Furthermore, the `Parliament` cluster offers reliable repeatable performance with little variance in execution time between runs. By providing a simple open specification, the `Parliament` cluster can offer these large speedups to other programming languages. Other languages that also do not support parallelism, such as Crystal [47] or Racket [48], can easily take advantage of data parallelism with little effort.

5.3 Achievements & Personal Remarks

This project began with me familiarising myself with distributed systems and the standard techniques taken by other solutions. Having never worked on distributed systems before it was important I understood the reasoning behind the design choices other systems had made, and the trade-offs they led to. I self-taught myself 2 new programming languages up to an advanced standard, including Rust, a language with a significant learning curve. I also successfully implemented a Consensus algorithm from a paper and increased my knowledge about containerisation.

5.4 Future Work

When designing and developing this project, there were several potential extensions I considered however due to the time limitations was not able to implement.

- **Extend support to other languages:** `Parliament` was built for OCaml, however due to its language agnostic communication, it could easily support another language by writing a new library for it.
- **Support for shuffling:** In standard MapReduce, between the map and reduce phase, there typically is a shuffle phase where data is transferred directly from the mappers to the reducers with the same key.
- **Data Locality:** Instead of constantly sending data back to the master, we can use data locality optimisations to schedule tasks on workers that already have the data.
- **Implement a policy-based scheduler**, currently the cluster allocates computation resources FIFO, however in a cluster with many users, a form of a policy for fairness could be more desirable.

Bibliography

[1] OCaml Programming Language. `https://ocaml.org`, May 2019.

[2] S. Kaisler and F. Armour and J. A. Espinosa and W. Money. Big Data: Issues and Challenges Moving Forward. In *2013 46th Hawaii International Conference on System Sciences*, pages 995–1004, Jan 2013.

[3] Ungar, David. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. *SIGPLAN Not.*, 19(5):157–167, April 1984.

[4] S. Dolan, L. White, A. Madhavapeddy. Multicore OCaml. Technical report, OCaml, 2014.

[5] Sutter, Herb. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[6] H. Esmaeilzadeh and E. Blem and R. S. Amant and K. Sankaralingam and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, June 2011.

[7] Rust Official Website. `https://www.rust-lang.org`, May 2019.

[8] Zaharia, Matei and Chowdhury, Mosharaf and Das, Tathagata and Dave, Ankur and Ma, Justin and McCauley, Murphy and Franklin, Michael J and Shenker, Scott and Stoica, Ion. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[9] Jean-Christophe Filliâtre, K. Kalyanasundaram. Functory: Distributed Computing for the Common Man. Technical report, LRI, December 2010.

[10] Mandel, Louis and Maranget, Luc. Programming in JoCaml (Tool Demonstration). In Drossopoulou, Sophia, editor, *Programming Languages and Systems*, pages 108–111, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[11] Cambridge Computer Science Tripos. `https://www.cl.cam.ac.uk/teaching`, May 2019.

[12] Marr, Deborah T. and Binns, Frank and Hill, David L. and Hinton, Glenn and Koufaty, David A. and Miller, J. Alan and Upton, Michael. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1):1, 02 2002.

[13] Blandy, Jim. *The Rust Programming Language: Fast, Safe, and Beautiful.* O'Reilly Media, Inc., 2015.

[14] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguade. How data volume affects spark based data analytics on a scale-up server. In Jianfeng Zhan, Rui Han, and Roberto V. Zicari, editors, *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, pages 81–92, Cham, 2016. Springer International Publishing.

[15] Rodgers, David P. Improvements in Multiprocessor System Design. *SIGARCH Comput. Archit. News*, 13(3):225–231, June 1985.

[16] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[17] Kennedy, Ken and McKinley, Kathryn S. Optimizing for Parallelism and Data Locality. In *Proceedings of the 6th International Conference on Supercomputing*, ICS '92, pages 323–334, New York, NY, USA, 1992. ACM.

[18] Kakadia, D. *Apache Mesos Essentials.* Packt Publishing, 2015.

[19] Aaron Davidson. Optimizing Shuffle Performance in Spark. 2013.

[20] Schneider, Fred B. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[21] Docker Official Website. `https://www.docker.com`, May 2019.

[22] Gantikow, Holger and Klingberg, Sebastian and Reich, Christoph. Container-based Virtualization for HPC. In *CLOSER*, pages 543–550, 2015.

[23] B. Long and P. Strooper. A case study in testing distributed systems. In *Proceedings 3rd International Symposium on Distributed Objects and Applications*, pages 20–29, Sep. 2001.

[24] Dowling, E. J. Testing Distributed Ada Programs. In *Proceedings of the Conference on Tri-Ada '89: Ada Technology in Context: Application, Development, and Deployment*, TRI-Ada '89, pages 517–527, New York, NY, USA, 1989. ACM.

[25] Jane Street Style Guide. `https://opensource.janestreet.com/standards`.

[26] Rust Style Guide. `https://github.com/rust-dev-tools/fmt-rfcs/blob/master/guide/guide.md`, May 2019.

[27] Travis Continuous Integration System. `https://travis-ci.com`, May 2019.

[28] ocamldoc for Module Marshal. `http://caml.inria.fr/pub/docs/manual-ocaml//libref/Marshal.html`.

[29] Michael, Maged M and Scott, Michael L. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. Technical report, Rochester University NY Department of Computer Science, 1995.

[30] Von Behren, Rob and Condit, Jeremy and Zhou, Feng and Necula, George C and Brewer, Eric. Capriccio: scalable threads for internet services. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 268–281. ACM, 2003.

[31] Bray, Tim. The Javascript Object Notation (JSON) Data interchange format. Technical report, 2017.

[32] Protocol Buffers - Google's data interchange format. `https://github.com/protocolbuffers/protobuf`.

[33] K. Maeda. Performance evaluation of object serialization libraries in XML, JSON and binary formats. In *2012 Second International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*, pages 177–182, May 2012.

[34] Lamport, Leslie and others. The part-time parliament. *ACM Transactions on Computer systems*, 16(2):133–169, 1998.

[35] Schneider, Fred B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[36] Lamport, Leslie. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[37] Vogels, Werner. Eventually Consistent. *Commununications. ACM*, 52(1):40–44, January 2009.

[38] Kubernetes Official Website. `https://kubernetes.io`, May 2019.

[39] etcd Official Website. `https://etcd.io/`, May 2019.

[40] Protobuf Compiler for OCaml. `https://github.com/mransan/ocaml-protoc`, May 2019.

[41] Rust implementation of Google protocol buffers. `https://github.com/stepancheg/rust-protobuf`, May 2019.

[42] S. Huang, J. Huang, J. Dai, T. Xie and B. Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51, March 2010.

[43] Spark Standalone Mode. `https://spark.apache.org/docs/latest/spark-standalone.html`.

[44] HiBench Suite Github. `https://github.com/intel-hadoop/HiBench`, May 2019.

[45] Robert Love. *Linux Kernel Development.* Sams, Second edition, 2005.

[46] Van Meter, Rodney. Observing the effects of multi-zone disks. In *Proceedings of the Usenix Technical Conference*, pages 19–30, 1997.

[47] Crystal Programming Language. `https://crystal-lang.org`, May 2019.

[48] Racket Programming Language. `https://racket-lang.org`, May 2019.

Appendix A

Communication Protobuf files/specification

A.1 Parliament → PM protobuf file

```
syntax = "proto3";

// USER -> MASTER
// Response: DataRetrievalResponse, ServerMessage
message DataRetrievalRequest {
    string user_id = 1;
    int32 job_id = 2;
}

// MASTER -> USER
// Request: DataRetrievalRequest
message DataRetrievalResponse {
    repeated bytes bytes = 1;
}

message InputAction {
    repeated bytes data_loc_in = 1;
    /*
        This is input data for the tasks in the first job.
    */
}

message MapAction {
    enum MapType {
        SINGLE_IN_VARIABLE_OUT = 0;
        SINGLE_IN_SINGLE_OUT = 1;
        VARIABLE_IN_SINGLE_OUT = 2;
    }
    MapType mapType = 1;
    int32 job_id_in = 2;
    bytes function_closure = 3;
}

message Job {
    int32 job_id = 1;
    /*
        Job id of the first job, needs to be unique to the user
    */

    oneof action {
        InputAction input = 4;
        MapAction map = 5;
    }
}

// USER -> MASTER
// Response: JobSubmissionResponse
message JobSubmission {
    string user_id = 1;
    repeated Job jobs = 2;
}

// MASTER -> USER
// Request: JobSubmission
message JobSubmissionResponse {
    bool job_accepted = 1;
    /*
        Indicates whether the job has been accepted by the cluster
    */
}

// USER -> MASTER
// Response: CreateConnectionResponse
message CreateConnectionRequest {
    string authentication = 1;
    string docker_name = 2;
    /*
        When running from Docker, the docker name needs to be passed into the
 ↪  cluster so the MemberOfParliaments can start the right container
    */
    string id_override = 3;
}

// MASTER -> USER
// Request: CreateConnectionRequest
message CreateConnectionResponse {
    string user_id = 1;
    /*
```

```
        This is the identifier given to user, which they use for further requests.
        This value is unspecified if connection_accepted = false
    */

    bool connection_accepted = 2;
    /*
        If true, the cluster has accepted the connection, however no requirement
↪   for the cluster to accept
    */
}

// USER -> MASTER
// Response: JobStatusResponse
message JobStatusRequest {
    string user_id = 1;
    repeated int32 job_ids = 2;
    /*
        Job Ids that the user wants information about
    */
}

// MASTER -> USER
// Request: JobStatusRequest, ServerMessage
message JobStatusResponse {
    message JobStatus {
        int32 job_id = 2;
        enum Status {
            BLOCKED = 0;
            QUEUED = 1;
            RUNNING = 2;
            COMPLETED = 4;
            HALTED = 5;
            CANCELLED = 6;
        }
        Status status = 3;
    }
    repeated JobStatus job_statuses = 1;
}

// USER -> MASTER
// Response: JobStatusResponse
message ConnectionRequest {
    string user_id = 1;
    enum Action {
        HEARTBEAT = 0;
        CLOSE_CONNECTION = 1;
    }
    Action action = 2;
```

```
}

// MASTER -> USER
// Request: ConnectionRequest
message ConnectionResponse {
    bool request_accepted  = 1;
}

message ServerMessage {
    enum Action {
        USER_TIMEOUT = 0;
        MISSING_JOBS = 1;
        INTERNAL_SERVER_ERROR = 2;
    }
    Action action = 1;
}

message SingleUserRequest {
    oneof request {
        CreateConnectionRequest create_connection_request = 1;
        ConnectionRequest connection_request = 2;
        JobSubmission job_submission = 3;
        DataRetrievalRequest data_retrieval_request = 4;
        JobStatusRequest job_status_request = 5;
    }
}

message SingleUserResponse {
    oneof response {
        CreateConnectionResponse create_connection_response = 1;
        JobSubmissionResponse job_submission_response = 2;
        DataRetrievalResponse data_retrieval_response = 3;
        JobStatusResponse job_status_response = 4;
        ConnectionResponse connection_response = 5;
        ServerMessage server_message = 6;
    }
}
```

A.2 MoP → Parliament Proto file

```
message WorkerInput {
    bytes function_closure = 1;
    enum MapType {
        SINGLE_IN_VARIABLE_OUT = 0;
        SINGLE_IN_SINGLE_OUT = 1;
        VARIABLE_IN_SINGLE_OUT = 2;
    }
    MapType map_type = 2;
    repeated bytes datapack = 3;
}

message WorkerOutput {
    repeated bytes datapacks = 3;
}
```

A.3 MoP → PM protobuf file

```
syntax = "proto3";

// WORKER -> MASTER
//Response: WorkerConnectionResponse
message WorkerConnectionRequest {
    int32 port = 3;
    string id_override = 4;
}

// MASTER -> WORKER
// Request: WorkerConnectionRequest
message WorkerConnectionResponse {
    string worker_id = 1;
    /*
        This is the identifier given to user, which they use for further requests.
        This value is unspecified if connection_accepted = false
    */
    bool connection_accepted = 2;
    /*
        If true, the cluster has accepted the connection, however no requirement
 ↪  for the cluster to accept
    */
}

// MASTER -> WORKER
// Response: WorkerHeartbeatResponse
message WorkerHeartbeatRequest {
    string worker_id = 1;
    /*
        We send the worker's ID to itself to ensure in the case of
 ↪  multiple-masters that they all have the same ID
    */
}

// WORKER -> MASTER
// Request: WorkerHeartbeatRequest, WorkerTaskSubmissionRequest,
 ↪  WorkerTaskCancellationRequest
message WorkerHeartbeatResponse {
    enum HeartbeatStatus {
        AWAITING_TASK = 0;
        PROCESSING_TASK = 1;
        HALTED_TASK = 2;
        CANCELLED_TASK = 3;
    }
    HeartbeatStatus status = 1;
    /*
        Send the status of the task we are currently processing
    */

    string task_id = 2;
    /*
        We send the task_id in the case of multiple-masters's eventual
    ↪   consistency, so the follower replicas can 'learn' about assigned tasks to
    ↪   workers
    */

}

// MASTER -> WORKER
// Responds with a WorkerHeartbeatResponse
message WorkerTaskSubmissionRequest {
    string worker_id = 1;
    /*
        We send the worker's ID to itself to ensure in the case of
    ↪   multiple-masters that they all have the same ID.
        Since it is the leader replica, it should always have the correct value
    ↪   for the worker_id
    */

    string task_id = 2;
    string docker_name = 3;
    repeated bytes data_in = 4;
    bytes closure = 5;
    enum MapType {
        SINGLE_IN_VARIABLE_OUT = 0;
        SINGLE_IN_SINGLE_OUT = 1;
        VARIABLE_IN_SINGLE_OUT = 2;
    }
    MapType map_type = 6;
    /*
        We send the map type, so that the output can be validated to be of a
    ↪   correct format
    */
}

// WORKER -> MASTER
// Response: WorkerFinishedResponse
```

```
message WorkerFinishedRequest {
    string worker_id = 1;
    string task_id = 2;

    enum WorkerTaskStatus {
        TASK_FINISHED = 0;
        TASK_ERRORED = 1;
    }
    WorkerTaskStatus status = 3;
    repeated bytes data_out = 4;
}

// MASTER -> WORKER
// Request: WorkerFinishedRequest
message WorkerFinishedResponse {
    bool response_processed = 1;
}

// MASTER -> WORKER
// Response: HeartbeatResponse
message WorkerTaskCancellationRequest {
    string worker_id = 1;
    /*
        We send the worker's ID to itself to ensure in the case of
↪   multiple-masters that they all have the same ID.
        Since it is the leader replica, it should always have the correct value
↪   for the worker_id
    */
}

message SingleWorkerMessage {
    oneof message {
        WorkerConnectionRequest connection_request = 1;
        WorkerHeartbeatResponse heartbeat_response = 2;
        WorkerFinishedRequest finished_request = 3;
    }
}

message SingleServerMessage {
    oneof message {
        WorkerConnectionResponse connection_response = 1;
        WorkerHeartbeatRequest heartbeat_request = 2;
        WorkerTaskSubmissionRequest submission_request = 3;
        WorkerFinishedResponse finished_response = 4;
        WorkerTaskCancellationRequest cancellation_request = 5;
    }
}
```

A.4 Consensus → PM protobuf file

```
syntax = "proto3";

// CONSENSUS -> MASTER
// Response: ConsensusResponse
message ConsensusRequest {
    enum Action {
        SET_ACTIVE = 0;
        SET_PASSIVE = 1;
        SHUTDOWN = 2;
    }
    Action action = 1;
}

// CONSENSUS -> MASTER
// Request: ConsensusRequest
message ConsensusResponse {
}

message SingleWorkerMessage {
    oneof message {
        WorkerConnectionRequest connection_request = 1;
        WorkerHeartbeatResponse heartbeat_response = 2;
        WorkerFinishedRequest finished_request = 3;
        ConsensusRequest consensus_request = 4;
    }
}

message SingleServerMessage {
    oneof message {
        WorkerConnectionResponse connection_response = 1;
        WorkerHeartbeatRequest heartbeat_request = 2;
        WorkerTaskSubmissionRequest submission_request = 3;
        WorkerFinishedResponse finished_response = 4;
        WorkerTaskCancellationRequest cancellation_request = 5;
        ConsensusResponse consensus_response = 6;
    }
}
```

Appendix B

Spark Evaluation Code

B.1 WordCount

```scala
package uk.co.cl.cam.doaa2.diss

import java.io.File

import org.apache.spark.{SparkConf, SparkContext}
import org.spark_project.guava.base.CharMatcher

import scala.collection.mutable.ListBuffer

object WordCount {
  def main(args: Array[String]): Unit = {
    val inputDir = args(0)
    val outputDir = args(1)

    val inputFiles = getListOfFiles(inputDir)

    val conf = new SparkConf().setAppName("wordCount")
    val sc = new SparkContext(conf)

    val threads = new ListBuffer[Thread]()

    for (inputFile <- inputFiles) {
      val thread = new Thread {
        override def run: Unit = {
          val input =  sc.textFile(inputFile.getPath)
          val words = input.flatMap(line => line.toLowerCase.split(" "))
          val filtered_words = words.filter(line =>
          ↪  CharMatcher.ASCII.matchesAllOf(line))
          val counts = filtered_words.map(word => (word, 1)).reduceByKey{case (x,
          ↪  y) => x + y}
          val arr = counts.coalesce(1)
          arr.saveAsTextFile(outputDir + "/" + inputFile.getName)
        }
      }
      thread.start()
      threads += thread
    }

    for (thread <- threads.toList) thread.join()
  }

  def getListOfFiles(dir: String):List[File] = {
    val d = new File(dir)
    if (d.exists && d.isDirectory) {
      d.listFiles.filter(_.isFile).toList
    } else {
      List[File]()
    }
  }
}
```

B.2 Sort

```
package uk.co.cl.cam.doaa2.diss

import java.io.File

import org.apache.spark._

import scala.collection.mutable.ListBuffer

object Sort {
    def main(args: Array[String]): Unit = {
        val inputDir = args(0)
        val outputDir = args(1)

        val inputFiles = getListOfFiles(inputDir)

        val conf = new SparkConf().setAppName("sort")
        val sc = new SparkContext(conf)

        val threads = new ListBuffer[Thread]()

        for (inputFile <- inputFiles) {
            val thread = new Thread {
            override def run: Unit = {
                val input =  sc.textFile(inputFile.getPath)
                val words = input.sortBy(t => t)
                val arr = words.coalesce(1)
                arr.saveAsTextFile(outputDir + "/" + inputFile.getName)
            }
```

```
                }
                thread.start()
                threads += thread
            }

            for (thread <- threads.toList) thread.join()
        }

        def getListOfFiles(dir: String):List[File] = {
            val d = new File(dir)
            if (d.exists && d.isDirectory) {
                d.listFiles.filter(_.isFile).toList
            } else {
                List[File]()
            }
        }
    }
```

B.3 Sleep

```
package uk.co.cl.cam.doaa2.diss

import org.apache.spark.{SparkConf, SparkContext}

object Sleep {
    def main(args: Array[String]): Unit = {
        val sleepCount = Integer.parseInt(args(0))


        val conf = new SparkConf().setAppName("sleep")
        val sc = new SparkContext(conf)

        val workload = sc.parallelize(1 to sleepCount, sleepCount).map(_=>
        ↪  Thread.sleep(1000L))
        workload.collect()
        sc.stop()
    }
}
```

Appendix C

Project Proposal

Computer Science Tripos – Part II – Project Proposal

A distributed general-purpose cluster-computing framework for OCaml

2381B, Jesus College
Originator: 2381B
19 October 2018

Project Supervisor: Dr J. Crowcroft
Director of Studies: Dr C. Mascolo
Project Overseers: Dr M. Kuhn and Dr E. Kalyvianaki and Dr N. Amin

Introduction and Description of Work

The Owl library is an emerging numerical library written OCaml. However, it is hindered by the fact that OCaml does not offer true parallelism OCaml threading implementation creates user threads, rather than kernel threads so only one function can be running at any one time. This means that Owl is severely handicapped when compared to other numerical libraries in other languages which can take advantage of the ability to run threads on multiple cores concurrently.
There are also cluster-computing frameworks that exist for other languages such as Apache Spark$^{\text{TM}}$ which allow applications to run on a processing cluster. However, for example Spark, requires you to use a specific framework to write these applications and require you to use JVM-based/Python programming languages which do not offer the static-typing and type inference offered by OCaml.
The aim of this project would be to create a general-purpose library where a processing cluster could be setup and used by user(s) wanting to run parallelised OCaml applications on a large networked computing resource. This will allow users to utilise their computing

infrastructure more efficiently and gain better performance by running their code truly parallelised on multiple cores and/or computing nodes.
The performance of such a system will be evaluated by comparing a problem written using the Owl, a typical parallelisable problem and compare the running time on a single node, compared to using the OCaml processing cluster. I plan to compare a classic Map-Reduce problem of finding the number of occurrences of words in a set of text files.

Starting Point

Currently my relevant theoretical background is:

- Part 1A CST Foundations of Computer Science - Interacting with the ML Programming Language, same language family as OCaml
- Part 1B CST Concurrent and Distributed Systems - MapReduce, Remote Method Invocation
- Part 1B CST Programming in C

I have practical programming experience using Kubernetes and Spark from a summer internship, similar and relevant technologies to what I'm building. For this project, I will write the library in OCaml, a language I have never used before, and hence will need to spend some time learning the language. I potentially may use C/C++, a language I've only been exposed to in the *Programming in C* course.

Work to be Done

Define an internal and external cluster API
Define an API and workflow for communication between the cluster master and the OCaml user library as well as the cluster master and any worker nodes. This API will include health checks and sending data and the marshaling technique and workflow for a new *function* (see next) to the cluster

Creating a worker library and application
Implement a worker library, for the user, in OCaml that will convert the marshaled data from the cluster master to the user code and return the resulting data back. This library requires the user to add the relevant stubs to their code before compilation. These will be compiled and submitted to the cluster as *'function'*. Will also need to implement a worker application which will run on the worker nodes and receive *functions* and data and execute on demand.

Creating a cluster master application
Design and develop a cluster master application that allows both several workers and users to connect simultaneously. The master will allowing the queuing of jobs submitted by users, as well as, monitoring the health of the workers and ensure no data or jobs are

lost. The master will also ensure the security of the data and *function* submitted by users, i.e. users can only access and use data/functions that they have added to the cluster.

Creating a user OCaml library
Design and develop an OCaml library to be used by users writing code to run on the processing cluster. This library will allow users to connect to a cluster, submit *functions*, and submit data for functions running on the processing cluster. The library will also handle the connection and the marshaling of data from the user's computer to the processing cluster.

Evaluation
Evaluation will involve benchmarking the performance of the Owl library running on a single node, compared to running on a processing cluster of multiple nodes with similar specifications to the original node. If I have time for any of the extensions, I can benchmark their performances, however the proposed extensions are focused on improving the reliability, scalability and security of the jobs on the cluster so, as a result, performance is likely to fall. It may also be interesting to compare the performance of the same algorithm running against an Apache Spark$^{\text{TM}}$ cluster on the same hardware.

Success Criterion

- Create a processing cluster for OCaml code to parallelise workloads
- Create an OCaml library that communicates with a processing cluster
- Reduce the running-time of a Map-Reduce program written in the Owl library, comparing single-node performance to new library running on multiple nodes.
  Evaluation Map-Reduce problem will be finding the number of occurrences of words in a set of text files.

Possible Extensions

Running the jobs in a containerised environment
Containerisation would involve encapsulating jobs in a container with its own application environment. This would introduce a software boundary between all the jobs running on a particular node. Some of the big advantages of this would be security and safety as the jobs would be isolated so they wouldn't be able to interact with each other Another benefit is environment consistency, so we can ensure jobs are reproducible, repeatable with a predictable application performance. Also, if and when, the cluster scales, it provides a blueprint for new workers for each new setup. Evaluation can be performed by creating a job that interacts in its environment in an harmful way; then checking that the other jobs on the system continue running as intended with no changes to their respective environment.

Run on a container orchestration system such as Kubernetes
Kubernetes is a platform for managing containerised workloads, which, as well as, the advantages provided by containerisation, kubernetes would provide a further abstraction from the actual physical node/virtual machine where the cluster master would just start a kubernetes pod (container) on the cluster without knowing/caring about the physical node that the container was running on. This would allow kubernetes to handle the node discovery failure, further decoupling the application from infrastructure.

Allow multiple cluster masters
Currently the architecture assumes 1 master and N workers, however this leads to a single point of failure as all of the data and functions passes through a single node. By employing distributed storage and processing techniques, all masters can be aware of each other and all of the jobs in all of the queues which will mean that if a master were to crash or become unreachable, another master could take over and return any results to the user, without the user(s) or workers(s) being aware of any failure. This would require some coordination between the masters for job scheduling as workers could become under contention when several masters have jobs in their respective queues. Evaluation of this can be performed when by creating at least 2 masters and 2 workers, and monitoring the job queue to ensure that there aren't any jobs being continually starved. Then causing a master node failure, and making sure all of the jobs that are still in execution / the queue are processed correctly and returned back to the user.

Timetable and Milestones

22nd October 2018 - 4th November 2018 (Preparation Work)

- Learning OCaml and socket communication between different OCaml processes on different nodes on the same network
- Evaluate whether OCaml is a suitable choice for the cluster orchestrator/worker application or to use a typical imperative language such as C/C++, due to the non-parallelism offered by OCaml
- Explore solution for queuing requests to the processing cluster
- Explore solution for caching requests efficiently

Milestones: Written a simple OCaml module for internal use for communication with the cluster

5th November 2018 - 18th November 2018 (Implementation)

- Build simple OCaml library to marshal data to and from the function to the worker implementation.
- Building the worker implementation which can receive functions and data from a socket, execute and return data reliably

- Add a simple command line interface for connecting the worker implementation to the cluster master

Milestones: Design and implement a simple API for communication between the cluster and the worker; use unit tests to mock a cluster master and mock a sample job being processed by the worker and test communication between OCaml worker library and the worker process.

19th November 2018 - 2nd December 2018 (Implementation)

- Building the cluster orchestrator for the user code to connect to.
- Add node discovery and recovery from node failure

Milestones: Build the main cluster orchestrator with simple ability to handle node unreliability; use unit tests to mock sample workers processing jobs

3rd December 2018 - 31th December 2018 (Implementation)

- Building the client library which will connect to cluster orchestrator
- Test initially whether code can run on multiple worker nodes

Milestones: I should be able to submit a sample job to the cluster using the library and it execute on separate worker nodes; achieve performance success criterion by comparing performance between single node and prototype processing cluster

1st January 2018 - 14th January 2018 (Implementation)

- Add caching of input and output for job restart in case of node failure/crash, mid-execution
- Add queuing to the cluster orchestrator to handle multiple requests from different users
- Add timeouts during jobs so that orchestrator can restart jobs, in case of node failure/crash
- Start writing progress report

Milestones: All core parts of the projects completed; first draft of the progress report completed

15th January 2018 - 31st January 2018 (Write Up)

- Finish any uncompleted tasks of the core project.
- Begin to work on extensions (Running the jobs in a containerised environment)
- Finalise and submit the progress report and presentation

Milestones: Progress report and presentation completed

1st February 2018 - 14th February 2018 (Write-Up)

- Produce data from project to provide quantifiable proof that success criterion has been met.
- Work on more extensions: Run on a container orchestration system such as Kubernetes, Allow multiple masters

Milestones: Have sufficient quantitive data produced from implementation achieving the success criteria

15th February 2018 - 15th March 2018 (Write-Up)

- Write first draft of the dissertation
- Proofread entire dissertation
- Submit completed draft to a number of individuals including my Director of Studies and Supervisor for feedback

Milestones: By the end of Lent Term, I should have finished my first draft of my dissertation giving me enough time to refine it and ask for feedback from multiple individuals

15th March 2018 - 23rd April 2018 (Write-Up)

- Revise draft and make changes based on feedback
- Preparation of code for upload
- Submit the final version of dissertation

Milestones: Dissertation complete

Resource Declaration

I will use my own laptop, an Apple MacBook Pro with a dual-core CPU and 8GB of RAM, running MacOS 10.12. I will use `git`, with a remote private repository provided by GitHub for version control as well as a Dropbox folder for backups. Should my laptop fail, I have a Microsoft Surface laptop at my disposal as well as using the MCS machines provided. I accept full responsibility for this machine and have adequate contingency plans to protect against hardware or software failure.