

Antanas Uršulis

Cluster-in-a-box

task parallel computing on many-core machines

Part II of the Computer Science Tripos

Queens' College

May 17, 2013

Proforma

Name: **Antanas Uršulis**
College: **Queens' College**
Project title: **Cluster-in-a-box: task parallel computing on many-core machines**
Examination: **Part II of the Computer Science Tripos, 2013**
Word count: **8865 words**
Project originator: **Antanas Uršulis and Malte Schwarzkopf**
Supervisor: **Malte Schwarzkopf**

Original aims of the project

Design, implement and evaluate an in-memory storage layer for the CIEL distributed task-parallel execution engine. The layer should support storing results of intermediate tasks without any disk backing. It should support both traditional and non-cache-coherent architectures, and make use of their special features, while limiting contention on shared hard drives. Finally, CIEL should be modified to use shared memory communication and the use of fine-grained locking in improving concurrency should be investigated.

Work completed

The above aims were completed by implementing a shared memory controller and a shared in-memory filesystem. These components were integrated with CIEL and supplemented with an I/O scheduler. The system was designed to run on traditional computers and the Intel Single-Chip Cloud. The work was made challenging by CIEL bugs and so implementing extensions was not considered.

Special difficulties

Many bugs in the original CIEL code base, that even the authors of the system could not figure out, caused considerable delays to the proposed schedule, mainly parts relating to evaluation.

Declaration of originality

I, Antanas Uršulis of Queens' 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. I give permission for my dissertation to be made available in the archive area of the Laboratory's website.

Signed

Date May 17, 2013

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Related work	2
1.3	Summary	3
2	Preparation	5
2.1	Introduction to CIEL	5
2.1.1	Tasks and task graphs	5
2.1.2	Executors	6
2.1.3	Reference fetching	7
2.2	Introduction to Intel’s SCC	8
2.2.1	No cache coherence	8
2.2.2	Message Passing Buffers	9
2.2.3	Shared disk access	9
2.3	Requirements analysis	9
2.4	Tools used	10
2.4.1	Programming languages	10
2.4.2	Libraries and standard modules	11
2.4.3	Choice of operating system	12
2.4.4	Development environments	12
2.4.5	Version control	13
2.4.6	Backup strategy	13
2.5	Software engineering techniques	13
2.6	Summary	13
3	Implementation	15
3.1	High-level architecture	15
3.2	Connecting CIEL with SHMD	17
3.2.1	Initialisation	17
3.2.2	Types of messages exchanged	17
3.2.3	The client	17
3.2.4	The server	18
3.3	Communication between SHMD instances	19

3.3.1	Types of messages exchanged	19
3.3.2	Core operation	19
3.3.3	Complications on the SCC	19
3.4	Shared memory workers	20
3.4.1	State objects	20
3.4.2	Recursive vs non-recursive requests	21
3.4.3	General operation	21
3.5	Shared memory files	21
3.5.1	Regular Linux	21
3.5.2	SCC Linux	22
3.6	I/O scheduling	25
3.7	Summary	25
4	Evaluation	27
4.1	Overview of results	27
4.2	Unit testing of filesystem	28
4.2.1	Preparation	28
4.2.2	Results	29
4.3	Testing machines	29
4.4	Benchmarking of I/O scheduler	30
4.4.1	Setup	30
4.4.2	Results	30
4.5	Further benchmarks	31
4.6	Summary	33
5	Conclusions	35
5.1	Achievements	35
5.2	Lessons learned	35
5.3	Future work	36
	Bibliography	37
A	shmfs unit testing script	41
B	I/O benchmarking script	45
C	Test generation for I/O benchmark	47
D	Project proposal	49

Chapter 1

Introduction

This dissertation presents a shared memory controller and filesystem that speeds up task-parallel computations on many-core computers. In-memory storage is a major topic in current datacentre computing research and it provides exciting performance improvements. My work describes the design, implementation, and evaluation of one such optimisation.

1.1 Motivation

Exploiting parallelism has been of great importance to computing since the field's early days. It is evidenced by the use of parallel techniques, such as pipelining and vector processing, in Computer Architecture, the use of multiple central processing units in both supercomputers and consumer hardware, and, more recently, the emergence of task-parallel execution engines, such as Google's MapReduce [5] or Cambridge's own CIEL [14]. The engines provide programmers with an easy-to-use framework which helps them implement distributed algorithms. These frameworks manage the often tedious and potentially gory details of distributing code and data across the network and dealing with faults. They were, however, designed at a time when commodity computers had only a few (1–4) processor cores, and thus they execute algorithms on a large number¹ of such computers within a datacentre network. However, looking at the present, we see machines with tens or hundreds of cores becoming prevalent. Extrapolating further, non-cache-coherent architectures, such as Intel's Single-Chip Cloud Computer (which I shall introduce in Section 2.2), could scale to thousands of cores with a suitable manufacturing process [2].

Such massively parallel computers bring various benefits to datacentre computing. First, fewer physical machines are required to provision a certain number of computation nodes. This in turn means the design is potentially more power-efficient, and also that sets of relatively small tasks can stay within the confines of a single machine, thus reducing network

¹To get an idea of exactly how enormous this number can become, I recommend watching Luiz Barroso's keynote on Warehouse-Scale Computing at Google [1]

traffic. Naturally, the same execution engines can be used with these many-core computers, but they are not optimised for this environment.

This project will try to look at the performance issues present in the CIEL engine when running on many-core machines, and solve some of them. A particularly interesting area for optimisation is the use of memory for storing files. This is, of course, due to memory being 80 times¹ faster than spinning-platter hard drives. Another reason why memory storage has become appealing is that the capacity now has become large enough to hold big data sets: it is not uncommon to find a personal computer with 8 GB of RAM, and servers with hundreds of gigabytes of RAM are certainly feasible, and can potentially perform better than equivalent clusters [18].

While implementing the memory caching layer, I also looked at how to limit the number of disk operations performed concurrently to eliminate poor seeking patterns and thus squeeze out more performance.

The work should make computations on many-core machines faster and also hopefully provide some insights on how to design execution engines for these environments.

1.2 Related work

Before moving on with the core of the work, we can briefly survey related work that was previously done, or is ongoing research in this area.

This dissertation is largely based on the “Condensing the Cloud” paper [19], which originally identified the problems that CIEL has when run in a many-core environment. The Firmament² execution engine, currently under development at the Computer Laboratory, aims to have in-memory caches for its objects. BigTable [3] is a distributed data store used internally at Google and it extensively uses in-memory caching to improve performance.

A very recent development, which was not yet available when this project started, is Tachyon³, developed by the AMPLab at the University of California, Berkeley. Tachyon is a distributed, fault tolerant filesystem for MapReduce, Hadoop, Spark and similar cluster frameworks, that aggressively uses memory caching. The Spark framework provides an abstraction called Resilient Distributed Datasets [21], which are cached in-memory and greatly speed up iterative computation tasks. Finally, RAMClouds [16] are a DRAM-based storage system for datacentres, proposed by researchers at Stanford University.

¹Estimate due to Peter Norvig [15]

²<http://www.cl.cam.ac.uk/~ms705/research/firmament/>

³<http://tachyon-project.org/>

1.3 Summary

Here is a quick overview of the rest of this dissertation. In Chapter 2, I will describe what was done in preparation for the project. Chapter 3 shall introduce the implementation: what was developed, how and why. Chapter 4 will give the qualitative and quantitative results of my work. Finally, Chapter 5 will draw out conclusions and provide pointers to future work.

Chapter 2

Preparation

Taking on this project required significant preparatory work. Since it involved modifying a complex system and working with new hardware, these had to be studied and understood. Also, a proper professional approach to the problem had to be developed, so that the project could go smoothly. In this chapter, I present the work I did before starting coding and my findings.

2.1 Introduction to Ciel

In this section, I shall introduce CIEL, elaborating on the parts of it relevant to this project, and present the problems that this project aims to solve.

CIEL is a distributed task-parallel execution engine, developed at the Systems Research Group of the University of Cambridge Computer Laboratory. Its job is to facilitate the development and execution of distributed algorithms on computer clusters, by providing a framework that manages distribution of tasks and data, scheduling, dealing with faults and retrieving results. Figure 2.1 displays an example use of CIEL. Now let us move on to explore some of CIEL's features in detail, see where the optimisations for many-core architectures lie, and how the design of CIEL will influence the design of the system.

2.1.1 Tasks and task graphs

Algorithms can be imagined to be composed of simple tasks, each having inputs and outputs, which are arranged in a certain order to produce a final result. This is in fact one common way of describing algorithms: a flowchart. Thus, generally speaking, task execution engines require their jobs (algorithms) to be specified in a similar way: by describing the individual tasks, their inputs and outputs, and the dependencies between them, therefore forming a data dependency graph. The engine can then use the graph to

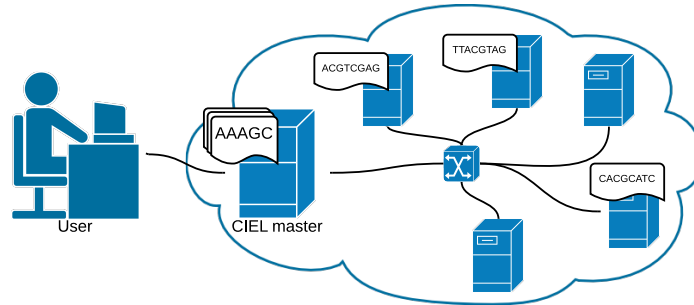


Figure 2.1: An example use of CIEL. The user, a biologist, submits to the cluster a DNA sequence alignment program along with some input data. CIEL manages the details of data distribution and later on returns the result.

infer the order in which to run the tasks, and parallelise accordingly.

One of the main contributions of CIEL to the field of distributed executors is the structure of its task graphs. Unlike MapReduce, whose task graphs are fixed and regular (composed simply of two stages, Map and Reduce), CIEL allows arbitrary data dependencies (so the graph is a DAG) and it also lets tasks spawn new tasks (making the graph dynamic). Looking at Figure 2.2, we can see that some tasks may produce outputs which are not returned as part of the final result. One might think it is not necessary to retain these intermediate outputs after they have been consumed by further tasks, yet CIEL usually writes every output to disk. This is done not only for simplicity, but also for fault tolerance: if the outputs are retained and some further task fails, the computation can be restarted with a minimal amount of work having to be redone. However, if we think about the case we are optimising for — a many-core computer — there is usually a single point of failure: for example, if the power supply fails, the whole system turns off. Therefore we can relax the fault tolerance requirement and *use nonpersistent storage*, i.e. main memory, for outputs, which should provide a performance improvement over hard drives.

2.1.2 Executors

So far we have talked about how tasks are arranged and executed, but not how they are themselves implemented. CIEL allows developers to write their tasks in any language, as long as the programs implement its interface, which involves exchanging JavaScript Object Notation (JSON) messages over a pair of named FIFOs residing in the filesystem and passed as arguments to the program. Libraries for Java, OCaml and C are provided that implement this interface and give the developer native functions which can, for example,

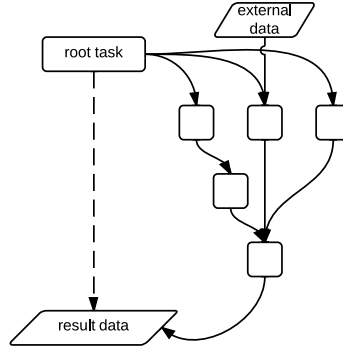


Figure 2.2: An example CIEL task graph.

spawn a new task. Such programs are run by the generic `proc` executor. CIEL also provides a language called Skywriting [13], which has its own executor. Executors are also available for interfacing with UNIX shell utilities which use standard input and output.

It is important to note that there are multiple executors and that tasks receive their inputs and outputs (data to be processed, not the interface FIFOs) as filenames or file descriptors. Therefore, any optimisations we make must not break the file abstraction provided to tasks if we do not want to make changes to several executors and also have to adopt the task programs to use the new method of accessing data. We can conclude that to implement the memory storage optimisation introduced in the previous section we shall require an *in-memory filesystem* of some sort. Of course, this is provided by tmpfs on the Linux operating system [17], but that will not be available across cores on the SCC computer discussed in Section 2.2.

2.1.3 Reference fetching

Now that we have established our core approach to the optimisation, we can look at how to make CIEL use our in-memory filesystem. Let us investigate how CIEL fetches references, which are a way of describing a current object or future object to be created.

Once a task is scheduled to be run, CIEL has to make sure every input is available locally. The executor calls into the *fetcher* module to retrieve the inputs. The result is a list of filenames that are guaranteed to exist and be readable. Internally, the fetcher sets up asynchronous fetch requests for every input, and for each request produces a *plan*: a list of methods that will attempt to fetch the file. A plan either produces a filename, or fails, in which case the next plan from the list is tried. Therefore, it is convenient for our implementation to provide a plan which performs the required shared memory setup and returns a filename. This also, in theory, means that heterogeneous CIEL clusters, created by joining multiple clusters-in-a-box into a network, are easy to provide as no external APIs change.

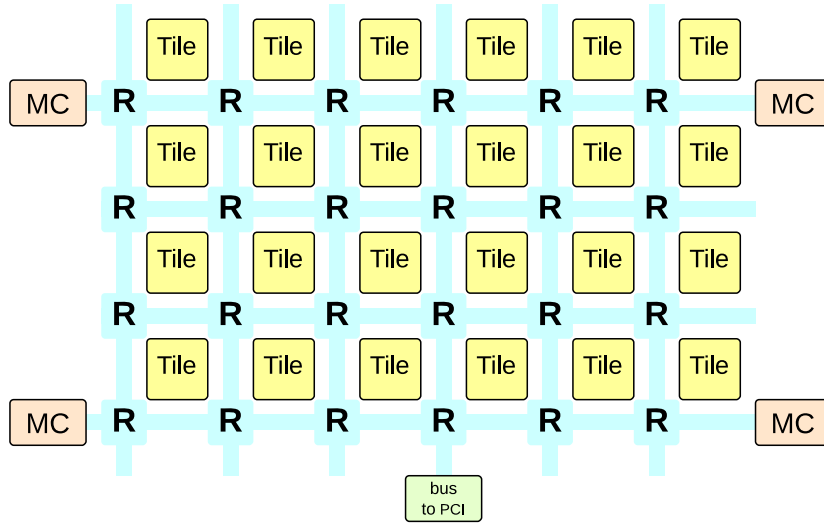


Figure 2.3: SCC chip layout, showing tiles, routers (*R*) and memory controllers (*MC*). Based on [12]

2.2 Introduction to Intel’s SCC

The Intel Single-Chip Cloud Computer is a 48-core processor for many-core computing research. It consists of 24 *tiles*, each having 2 modified second generation Pentium (P54C) cores, L2 caches and a router which connects the tile to a 2D mesh network-on-chip, as displayed in Figure 2.3. Each tile also includes a 16 KB Message Passing Buffer, and four DDR3 memory controllers connected to the network provide up to 64 GB of system memory. The following sections overview a few special features of this architecture and the implications it will have for our design.

2.2.1 No cache coherence

Typical multicore processors are cache-coherent, meaning that when a core performs a write to a cache line, this update is propagated to other cores that also have the same line in their caches, so that every core has the same view of memory contents. This is usually achieved by broadcasting a *cache-invalidate* message on the shared system bus, therefore all-to-all connectivity between cores is required.

As the number of cores integrated into a processor design grows, so does the complexity of the protocol required to maintain a consistent view of memory between their caches. Eventually, this protocol overhead would exceed the benefit of integrating more cores, so the SCC processor goes in a different direction by not maintaining cache coherence. This however means that conventional operating systems like Linux cannot operate with a single image across all cores. Instead, in the usual configuration, this computer partitions

its memory into private blocks and boots a separate instance of Linux on every core. Therefore, the result is much like a traditional distributed system, a cluster-on-chip. So our filesystem, when running on the SCC, will have to allow concurrent access from any core to the noncacheable shared memory that is available to cores.

2.2.2 Message Passing Buffers

In order to enable low latency communication between cores, the SCC introduced Message Passing Buffers. These are small (8 KB per core) SRAM memories, visible to the programmer as a single shared address space, with cache coherence maintained in software by special instructions which were added to the ISA. The MPBs can be used directly, although it is more convenient to use the RCCE library (Section 2.4.2), which implements an MPI¹-like interface. In other words, cores can deposit messages for other cores in the MPBs, and these will appear to the other cores with only a few cycles' delay. We shall make use of these buffers for communication between instances of our filesystem controller.

2.2.3 Shared disk access

Every Linux instance has access to a single shared NFS mount, stored on the SCC's management console. It is also highly likely that future many-core architectures would have fewer disks than cores. Therefore, concurrently scheduling multiple tasks that read or write to a single hard drive might cause seeking patterns that degrade performance. Our filesystem would benefit from including some sort of scheduler that limits I/O contention.

2.3 Requirements analysis

In this section I shall present what requirements were derived for the project from the material learnt about CIEL and the Single-Chip Cloud computer.

The first thing I noted in Section 2.1.1 is that fault-tolerance restrictions can be relaxed and memory can be used as storage. Therefore, *my system will have to support both input and output to memory.*

In Section 2.1.2 I made the observation that there exist many executors for CIEL, and they supply inputs to tasks as *files*. So, *my system will have to provide a file abstraction to memory storage*, and this effectively means *I shall provide an in-memory filesystem.*

Because the Single-Chip Cloud computer does not have cache coherence and thus behaves like a distributed system, *I will have to use message passing techniques throughout the system.*

¹Message Passing Interface standard [6]

Finally, because concurrent access from many cores to a shared disk will cause unwanted contention, *I will have to provide a means of limiting the amount of concurrent I/O jobs.*

2.4 Tools used

2.4.1 Programming languages

C

The core of the project, a shared memory control daemon, `shmd`, was chosen to be implemented in C, mainly for two reasons:

1. The RCCE and iRCCE libraries for manipulating SCC’s architectural features are written in C, therefore we can interface them directly.
2. The C language provides low-level access to memory, which is necessary to implement the shared memory filesystem, as we shall see in Section 3.5.2.

In addition to that, familiarity with programming in C and C++ was another factor: I had attended many programming contests in high-school that used C/C++. In addition, I worked as an intern at the Computer Laboratory’s Systems Research Group over the summer of 2012, where I reimplemented parts of the “Building an Internet Router” course in C++. However, I did have to familiarise myself with a lot of the standard POSIX functions and IPC mechanisms, like pipes, named FIFOs and `select()` loops.

Python

For my project, I had to modify parts of CIEL to use the newly written in-memory store code. Only about 50 lines of Python code were written, but significant parts of the 9,000-line CIEL codebase had to be read and understood beforehand. Even though PEP 20, a document² describing the philosophy of the Python language, states that “Readability counts.”, I would argue that Python’s dynamic typing, combined with lack of documentation in the relevant parts of CIEL pose a challenge in trying to navigate and understand the code.

Other

Some shell scripting in Bash was used to automate the testing and evaluation of the project. A bit of familiarity with Java and Skywriting³ was required to run and tweak algorithm

²Reproducible by typing `import this` into the standard Python interpreter

³A language designed for use with CIEL: <https://github.com/mrry/ciel-skywriting>

examples used in evaluation.

2.4.2 Libraries and standard modules

RCCE

RCCE⁴ is a message passing library for the SCC computer, which uses its Message Passing Buffers for communication between cores. It provides blocking `send()` calls which must be matched by a corresponding `recv()` at the other end. This is a problem if one cannot predict the pattern in which cores will exchange messages, for if two cores try to `send()` to one another, they will deadlock. We shall see how the iRCCE extension, described later, solves this.

The library also provides mutual exclusion locks based on the test-and-set registers available on the chip. Furthermore, it implements barriers, which are a synchronisation method: a process waits at the barrier call until every process participating in the computation has reached it.

Finally, it allows allocating memory that is shared between all cores. Because of the method used [10], the amount is limited to 960 MB. However, my experiments have shown that the first 256 MB of it are unusable, as performing a write would take up to four cores offline. It remains unknown whether this was a configuration or hardware issue.

Source code⁵ of the library is available under the Apache License, version 2.0 and the developers have also written a relatively complete and up-to-date specification [11].

iRCCE

The iRCCE [4] library extends RCCE with non-blocking (asynchronous) message passing functions. They work by creating request handles, which then have to be continuously tested for completion. The function calls return immediately if a full transfer cannot be performed at the moment. Another contribution of the extension are source wildcards, which enable to perform a `recv()` from any core, not just the one specified in the call. We shall see how all this functionality is used in Section 3.3.3.

ctypes

The Python standard library includes the `ctypes` module, which allows to interface Python code with external C libraries. This is used to connect CIEL with a client library that talks to the `shmd`, as explained in Section 3.2. Overall, `ctypes` is extremely easy to use, as

⁴Pronounced “rocky”

⁵<http://marcbug.scc-dc.com/svn/repository/trunk/rcce/>

```
# Import the module
import ctypes

# Load the standard C library
libc = ctypes.cdll.LoadLibrary("libc.so.6")

# Prepare a mutable character array, as strings in Python are immutable
buf = ctypes.create_string_buffer(10)

# Perform formatted output into the string
libc.sprintf(buf, "Hello %d", 42)

# Print "Hello 42"
print buf.value
```

Figure 2.4: An example use of *ctypes*.

Figure 2.4 shows in an example. This allowed me to write less Python code and avoided having to deal with preparing properly aligned `structs` and passing them over sockets.

2.4.3 Choice of operating system

The project was developed and the dissertation written on my personal laptop, running Gentoo Linux. The coding, however, was done inside a virtual machine running Ubuntu Linux 10.04. This choice was made after some trial-and-error in getting CIEL to work first on Gentoo, then on Ubuntu 12.04. The reason why CIEL did not work with newer operating systems is most probably due to changes in one of its dependencies. While I have not tracked down the exact cause, I suspect there is a race condition somewhere in the pycURL fetcher, which then randomly results in deadlock.

2.4.4 Development environments

All of the C code was written with the Vim and gVim text editors. Vim (standing for “Vi IMproved”) is a modal text editor owing its roots to the `vi` editor created in 1976 for the UNIX operating system. It provides syntax highlighting for most programming languages and has great search-and-replace facilities as well as being controlled completely through a text interface, meaning one does not have to use the mouse.

For Python code, however, the Eclipse IDE, version 3.5.2, with the pyDev extension was used. Eclipse has navigation capabilities that are superior to those that Vim provides without plugins, and this was most important. The ability to find all occurrences of a class or function, or quickly jump to the definition of a variable were all extremely helpful in understanding CIEL’s code.

2.4.5 Version control

The original CIEL code, along with bindings and examples for various languages, is hosted on GitHub⁶. Thus it was natural to also use the Git version control system for this project. This allowed to work on the project from different machines (for example, submit changes while debugging on the SCC) and provided for backup, which I shall talk about in the next section.

2.4.6 Backup strategy

Because computer hardware can fail and accidents happen, a plan was developed to ensure that work will not be lost and can be resumed after disasters.

The master copy of the code was stored in a Git repository on GitHub, forked from the original CIEL code base. However, since it would not be wise to rely on a free but commercial service being available, the repository was also mirrored to the PWF and Emerald, a personal server away from Cambridge. This was done with a single command on a per-commit basis. The text of the dissertation was also stored in a Git repository, but only mirrored to the PWF and Emerald.

Also, regular but manual backups were done to an external hard drive. In case of my laptop failing, I could have resumed the work on one of the plenty PWF machines available.

2.5 Software engineering techniques

An iterative model of development was employed for this project. This allowed to gradually implement and test features, discovering important peculiarities of how CIEL works and adjusting my design along the way. The first version that was developed used POSIX shared memory (Section 3.5.1) and did not have an interdaemon server (Section 3.3), which was added on for the second version, refining other components as well. For the third iteration the `shmfs` (Section 3.5.2) was prototyped for use on regular Linux instances. Finally for the fourth iteration everything was ported to the SCC and the I/O scheduler (Section 3.6) was added.

2.6 Summary

This chapter gave an overview of the preparatory work done for the project. The CIEL distributed task-parallel execution engine and the Intel Single-Chip Cloud computer have

⁶<https://github.com/mrry/ciel>

been introduced, as well as fundamental requirements of my project. The tools and development methodology were also described, and the backup strategy explained.

Chapter 3

Implementation

In this chapter, I shall present the implementation work done for this project. First, we will see a high-level overview of what the components are and how they fit together, and then look at the implementation of each component. I will highlight important design choices, the difficulties I encountered, and how they were resolved.

3.1 High-level architecture

In Figure 3.1, we can see all of the components that comprise our system and the connections between them. Here I shall list them, stating briefly their purpose:

1. The SHMD client library, `libshmdc.so`, is an interface to SHMD.
2. CIEL includes a ctypes wrapper around the library, and together they form a *client*.
3. The local IPC server services requests from one or more clients over sockets.
4. The interdaemon IPC server services requests from other SHMDs via sockets on regular Linux, and message passing on the SCC.
5. Shared memory worker threads, created by the IPC servers, process requests.
6. The filesystem implementation is an interface to shared memory used by the workers.
7. A simple I/O limiting scheduler is included in the filesystem to reduce contention.

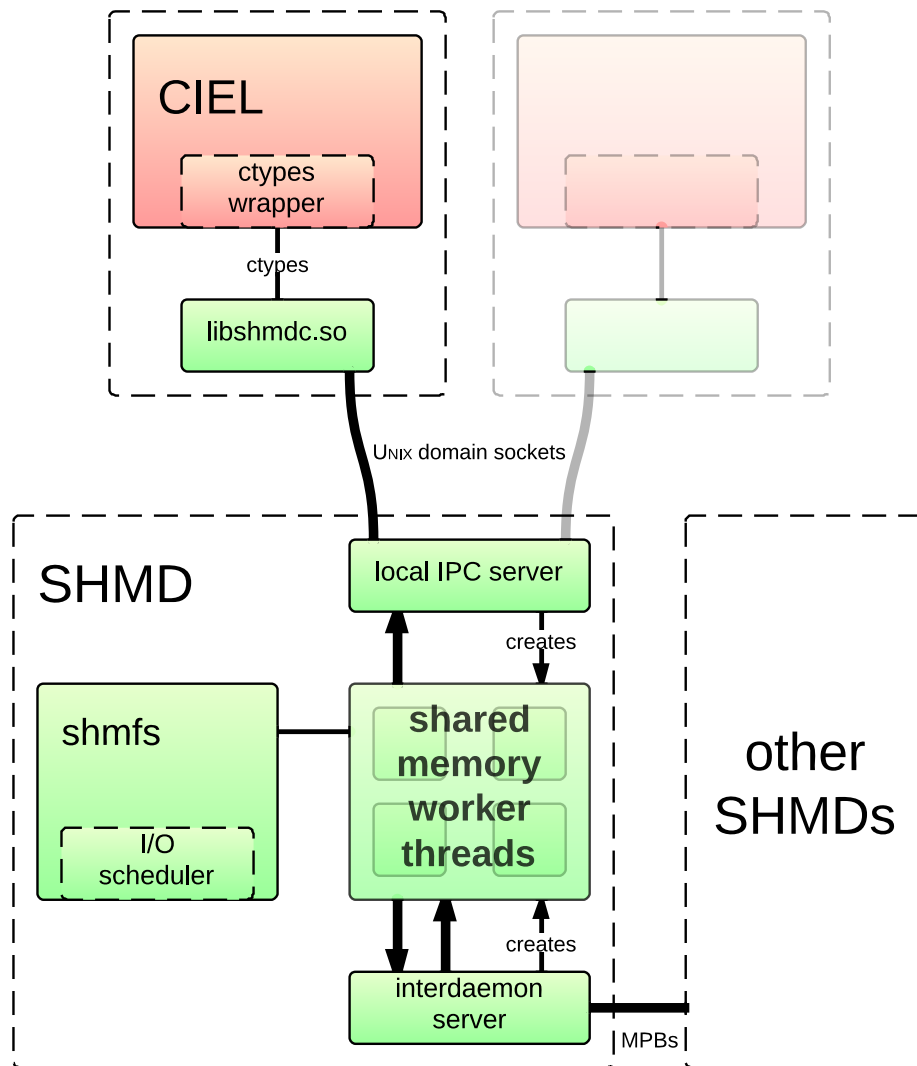


Figure 3.1: Components of the SHMD system, and their interactions. Red parts are implemented in Python, green ones in C. Thick arrows denote pipes.

3.2 Connecting CIEL with SHMD

Communication between CIEL and SHMD happens over UNIX domain sockets, a local interprocess communication mechanism. This section elaborates on important implementation details of this component.

3.2.1 Initialisation

UNIX domain sockets identify communication endpoints using names in the filesystem. Therefore, some common filenames have to be agreed upon if the client (CIEL) and server (SHMD) are to find one another. The server creates its socket in the on-disk blockstore’s directory, allowing one SHMD instance per blockstore. Every client places its sockets in a temporary directory created upon startup, and of course it needs to know the path to the blockstore, so that SHMD can be found.

3.2.2 Types of messages exchanged

The messages in Table 3.1, expressed as C `structs`, are sent over local IPC between the client library `libshmdc` and the SHMD.

Type	Sent by	Arguments
load request	client	pathname which CIEL wants to open for reading
write request	client	pathname which CIEL wants to open for writing
commit request	client	pathname which CIEL wants to commit, desired new name for the file
response: OK	server	pathname that CIEL should use
response: fail	server	failing pathname

Table 3.1: Message types in the local IPC protocol

3.2.3 The client

The client is implemented as a C library with a ctypes Python wrapper. Load request messages are sent from a new reference fetching “plan” (Section 2.1.3), which mostly mimics the one used for local files. Figure 3.2 illustrates one possible list of plans, and how my shared memory plan takes priority.

Write request messages were, however, trickier to implement. The obvious place to inject our own SHMD filenames was CIEL’s `producer_filename` function. It turned out that this

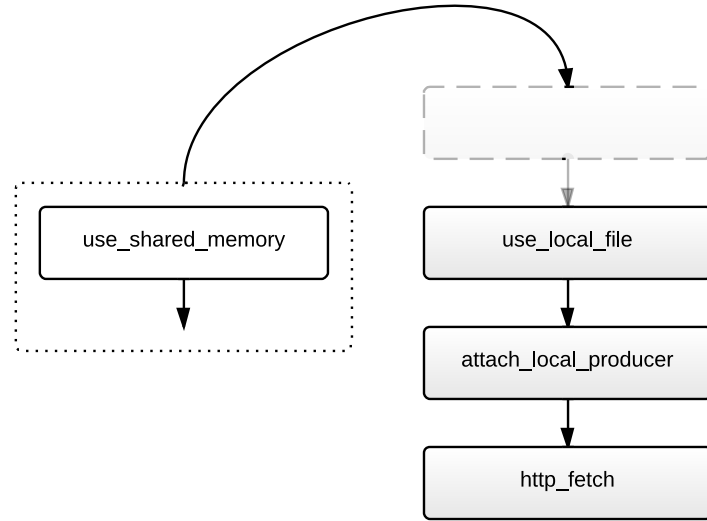


Figure 3.2: A plan list in CIEL’s fetcher module, and my modification.

function gets called more than once for the same reference ID. Sending repetitive requests to SHMD would be wasteful, if not troublesome (we shall see in Section 3.5.2 that FIFOs are created in response to requests). Therefore, I added a cache that remembers whether there was a request for this ID already, returning a cached SHMD response if there was, and performing a write request if there wasn’t. The effect of this is that even though `producer_filename` might be called multiple times, the file will be opened for writing only once.

Commit requests are sent when the CIEL worker has successfully produced an output and wishes to rename it from a temporary name to a permanent one.

3.2.4 The server

The server runs in its own thread within SHMD. Upon receiving a request, the server saves it, along with the sender’s address, into a *shared memory worker state* object (see Section 3.4) and spawns a new worker thread, in recursive mode, for this object before returning to wait for new requests. It also listens on a pipe which is included within the state object, so that the worker thread can signal back to the server by writing the address of its state object once it is done. Pipes were chosen because they are represented as file descriptors, thus can be used within a `select()` loop, unlike condition variables. Reusing the same socket IPC mechanism as for the CIEL-SHMD communication seemed a heavyweight alternative, and would have complicated the protocol.

3.3 Communication between SHMD instances

Whether running multiple blockstores on the same Linux instance, or a distinct blockstore for every core in the SCC, the SHMD instances for these stores will have to communicate. This is the responsibility of the *inter-daemon* server, whose role it is to process requests for files that are not present in the blockstore of the local SHMD. The server also runs in its own thread and has a high-level structure similar to the CIEL-SHMD bridge. This section describes how they differ.

3.3.1 Types of messages exchanged

Within the *inter-daemon* server, the “load request”, “response: OK” and “response: fail” messages from Table 3.1 are used, with the following addition: request messages also contain an integer identifying the shared memory worker that produced this request. The same integer must be included in the corresponding response messages.

3.3.2 Core operation

Upon receiving a load request from another SHMD, the server spawns a worker thread, this time in *non-recursive* mode. When the server receives a pointer to a worker state object on a pipe from a worker thread, it checks which worker stage is set in that object. If this shows that the worker is making a response, then this response is forwarded to the appropriate SHMD. If the worker state shows a recursive request, then a load request is broadcast to all the other SHMDs.

SHMDs are identified by an ID supplied manually on normal Linux, or by the core ID on the SCC. The total number of SHMDs is fixed upon startup as a command line parameter and cannot vary over time. Dynamic cluster membership should not be difficult to add, but it cannot be supported on the SCC due to limitations in the RCCE library.

3.3.3 Complications on the SCC

On regular Linux I use a separate UNIX domain socket for inter-SHMD communication. Of course, these are not available across kernel boundaries, so something different has to be used on the Single-Chip Cloud. TCP sockets could be used, but they have a high latency overhead for local communication [20]. Instead I employ the low-latency Message Passing Buffers. Unfortunately, the RCCE and iRCCE interfaces to the buffers do not provide a file descriptor abstraction, so I could not use a `select()` loop to wait for a message to arrive. But I also cannot simply block on a `recv()` call, because then the *inter-daemon* server might block forever, thereby also starving other requests, for example from the worker

```

Loop forever:
  Setup a receive request if one is not pending
  If neither broadcast nor send request is pending:
    Try reading from the pipe
    If successful:
      If recursive request, setup a broadcast request
      If response, setup a single send request

  If a receive request is pending, check its status:
    If done:
      Create a worker thread for the received message
      Release the request

  If a broadcast is pending, collect all statuses:
    If every request is done:
      Release the broadcast request

  If a single send is pending, check its status:
    If done:
      Release the request

  Sleep 1/10th of a second

```

Figure 3.3: The inter-daemon server loop on the SCC. Each status check pushes the corresponding request queues.

thread reply pipe. Therefore, I used iRCCE’s asynchronous message passing capabilities and perform a busy spin loop. The algorithm is presented in Figure 3.3.

3.4 Shared memory workers

Worker threads are started from the communication components to serve the requests that they receive. By offloading this work to separate threads I allow the servers to immediately continue to listen for requests. This is important as some of the tasks that a worker might perform potentially take a lot of time (consider loading a few-hundred megabyte file into memory). In the following, I describe the operation modes and behaviour of such worker threads.

3.4.1 State objects

When a worker thread is started, it is supplied with a state object which encapsulates everything about the request made, including its source. The state includes a pipe to which the worker will write its state address once it has finished serving the request. Finally, it includes a flag denoting whether the worker was started in recursive mode, and a stage indicator, which shows what operation the worker is currently performing: *i*) serving a request, *ii*) making a recursive request, or *iii*) responding.

3.4.2 Recursive vs non-recursive requests

When started in recursive mode, the worker is allowed to ask other SHMDs to service a request if it, for some reason, cannot do so itself. Currently, the only example of this is when a worker is requested to load a file that is not present in the local blockstore.

Requests coming from the CIEL-SHMD IPC server are started as recursive, while those received from other SHMDs are not. This prevents a snowball effect where a single recursive request generates new requests from every SHMD, leading to an exponential growth of requests in the system.

Recursive requests are issued by setting the appropriate stage in the state object, providing a pipe for replies, and writing the state address to a globally available pipe, which is connected to the inter-daemon server.

3.4.3 General operation

The first operation of a newly started worker thread is demultiplexing of the request header. This determines the type of request: write and commit requests are passed on to the filesystem component, load requests require a little more work.

For load requests, the worker checks whether the file already exists in shared memory and prepares it for reading if so. If not, the local filesystem is checked and a load-and-open call is issued to the shared memory filesystem if the file is present. If it is not present and the worker is operating in recursive mode, the inter-daemon server is instructed to broadcast the request to other SHMDs. Replies from other SHMDs are collected and the file is prepared for reading if at least one “response: OK” message is received. Otherwise, the worker reports that it has failed.

3.5 Shared memory files

This section describes how the shared memory filesystems were implemented on two different platforms.

3.5.1 Regular Linux

The POSIX.1-2001 [9] standard defines a shared memory object interface. On Linux, such objects are typically created in a tmpfs (in-memory) filesystem, mounted at `/dev/shm/`. I make use of this fact to implement the in-memory filesystem when all SHMDs are running under a single Linux kernel image.

My filesystem layer will therefore translate pathnames in the CIEL blockstore to names

under `/dev/shm/` and perform these additional actions. On a load request, it will invoke the standard `cp` utility to copy the file from hard disk to memory. On a commit request, it will invoke `ln` to create a hardlink in the `shm` filesystem.

3.5.2 SCC Linux

Let us recall that the SCC runs a separate copy of Linux on each core. Consequently, the convenient POSIX interface cannot be used to maintain a shared filespace between these instances. This section describes a custom filesystem that I implemented to provide such a space.

Filesystem structure

A fixed structure was chosen to simplify the design. The shared memory filesystem, let us call it `shmfs`, is a memory-mapped `struct` consisting of the following:

1. Metadata/statistics: number of free directory entries, inodes and blocks, number of I/O operations currently happening between `shmfs` and the local filesystem.
2. Directory: a predefined number of filename \rightarrow inode mappings, invalid entries are specified by a special inode ID. Only a flat namespace (no nested directories) is provided.
3. Inode table: a predefined number of index nodes, each containing:
 - the number of directory entries for the inode,
 - the number of handles currently open,
 - the number of the first block of data,
 - the size of the file represented by the inode,
 - some flags: is the inode valid, is it open for writing, has it been committed.
4. File Allocation Table: a map of all the data blocks in the `shmfs`.
5. Data blocks: these hold the actual data of files.

Separating the directory from the inode table allows to implement hardlinks, which are used for commits. Parameters such as block count and size, number of directory entries or inodes, are configurable at compile time.

Supported operations

Let us list the operations supported by **shmfs** and highlight a few important points.

create(name, openwrite) creates a new directory entry, allocates an inode and a block; the file can optionally be marked as open for writing.

lookup(name) searches the directory for the given name, returning an inode number if successful and a special value otherwise.

link(target, name) looks up the target, and creates a new directory entry with the name, pointing to the target's inode.

commit(target, name) links the name to the target and also sets the committed flag; a committed file is read-only.

load(name) copies from the local filesystem into **shmfs** and commits the file given.

store(inode, name) writes out the file given by the inode into the local filesystem.

FAT operation

The File Allocation Table in **shmfs** operates much like in the widely used FAT family (FAT16, FAT32, etc.) of filesystems for Microsoft DOS and Windows. Below is a short summary, also illustrated by Figure 3.4.

The inode of a file contains the number of the first block in the file. This is also a pointer into the file allocation table, where we store the number of the next block in the file. Special numbers are reserved to mark that a block is free or that it is the last block of a file. Therefore the numbers form a forward linked list in the FAT, which also corresponds to the list of blocks that contain the file's data.

Input/output FIFOs

So far **shmfs**, as described, does not provide a file abstraction in the local filesystem and thus cannot be used with CIEL. This part shows how we connect them.

Two pathname translation functions are provided: one for reading and one for writing. Each of them opens the file in **shmfs**, creating it if required, creates a named FIFO in the local filesystem, and spawns a thread that will service this FIFO. The functions return the name of the FIFO. So now, when a CIEL task opens the FIFO, it will be talking directly to our filesystem. The serving thread will either push data from **shmfs** or read into **shmfs**, depending on the type of FIFO opened.

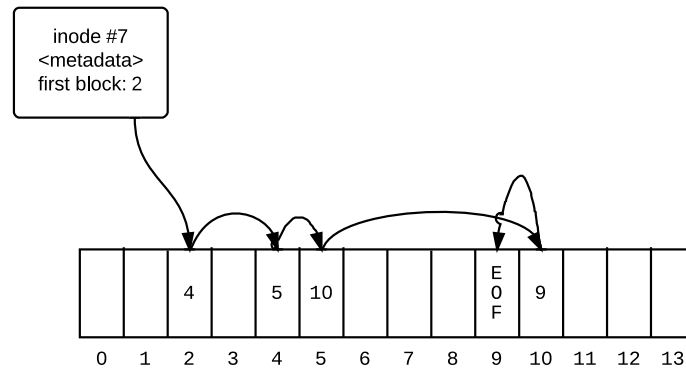


Figure 3.4: A visualisation of a linked list inside a FAT. The first block is acquired from the inode. A special value denotes end-of-file.

End-of-file is denoted by closing the pipe. Once either party closes the pipe, the serving thread removes the named FIFO from the filesystem, adjusts the handle count on the **shmfs** file, and terminates.

Locking behaviour

Remember that **shmfs** has to be accessed concurrently. Here we describe how that is achieved safely.

There are four separate locks: for statistics, the directory, inode table, and the FAT. Any operation that needs to read or write these sections of **shmfs** have to take out the appropriate locks. Locks are taken out according to a two-phase locking policy: *i*) a lock associated with an object must be taken before the object is used; *ii*) in the first phase, locks are taken out, but none are released; *iii*) in the second phase, locks are released, but none are taken. An additional constraint is that I impose a total order on locks to avoid deadlock. The following exceptions are made:

- There are no locks on the data portion of **shmfs**. This is justified by the assumptions that there will never be two or more writers to the file, and that there will never be a concurrent writer and reader.
- Readers do not require a lock on the FAT, even though they use it. This is also justified by assuming that concurrent reads and writes to the same file will not happen.

The assumptions above are a consequence of CIEL’s deterministic naming of objects, and the need to “commit” objects before they can be used.

A possibly better alternative to having just four locks would have been fine-grained locking. Greater concurrency can be achieved if there is a separate lock for every directory entry, inode, etc. However, the RCCE library only provides 48 locks and thus a fine-grained locking scheme may quickly run out of available locks to use.

3.6 I/O scheduling

Once `shmfs` is in place, we can easily track the number of I/O jobs that are accessing the shared hard drive backing `shmfs`. This allows us to enforce a policy that limits the number of such jobs running concurrently. In the current implementation, when an operation requiring disk I/O is requested (a file load into `shmfs`, for example), this number is checked, and if it exceeds some preset threshold (currently 1), then the requesting thread is put to sleep for some amount of time (currently $\frac{1}{10}$ s). When it wakes up, the number of I/O jobs is rechecked, and the thread may either proceed or sleep and recheck again, until it is finally allowed to perform the operation. Ideally, instead of having to sleep, I would like to have an interrupt-driven wait, so that a thread would only recheck the I/O job count when it is actually changed. This was, however, impossible to implement with the current RCCE and iRCCE libraries due to their complete lack of interrupt-driven methods. Finally, another alternative would have been to try and schedule a different CIEL task, preferably CPU-bound, while I/O limits are exceeded. However, adding awareness of CPU and I/O requirements to CIEL, along with an appropriate scheduler was beyond the scope of this project.

3.7 Summary

The chapter introduced the implementation part of the project. We saw an architectural overview of the shared memory controller daemon and then dove in to see the details of how it communicates with both CIEL and its sibling daemons and how it manages shared memory. A major part of the discussion was the in-memory filesystem developed for use on the Single-Chip Cloud computer. Finally, a simple I/O limiting scheduler was presented.

Chapter 4

Evaluation

This chapter presents the results of my work for the Cluster-in-a-box project. Qualitative and quantitative aspects are considered, and these are related to the success criteria from the original proposal.

4.1 Overview of results

The original project proposal (which is at the end of this document) set a number of criteria that had to be met in order for it to be deemed successful. Let me consider each one and summarise what was done:

1. *Have modified a task-parallel data flow execution engine to share its object store between instances and transfer intermediate results via shared memory.*

The SHMD, along with its in-memory filesystem implementations on both regular Linux (Section 3.5.1) and the SCC (Section 3.5.2), provides a shared object store and it is used for producer results as well.

2. *Have modified the engine to use shared memory instead of network-based communication.*

While some messages are still exchanged through the HTTP-based protocol, the majority of communication between CIEL instances (in terms of bytes) now happens via shared memory.

3. *Implemented at least one architecture-specific optimisation.*

On Intel's Single-Chip Cloud computer, I use the available shared off-chip memory, and also exploit the on-chip Message Passing Buffers for communication between SHMD instances, as detailed in Section 3.3.3.

4. *Implemented a scheduler that limits the number of tasks perform I/O concurrently.*

I described the approach taken to tackle this goal in Section 3.6.

5. *Investigated use of fine-grained locking to improve concurrency.*

Locks had to be used in the SHMD to secure access to the in-memory filesystem. It was found that the RCCE library does not provide enough locks to support a fine-grained approach.

6. *Compared the performance of various large-scale computations on the optimised and network-based variants of the engine.*

This is the topic of the rest of the chapter, specifically Sections 4.4 and 4.5.

Therefore I can conclude that the project was overall successful. Extensions were not considered due to delays caused by bugs in CIEL and/or its dependencies, and a lack of working example programs for CIEL. However, I can briefly mention that the following proposed extension:

Tasks can be scheduled on different cores within a single machine, and also sent to a different machine across the network.

should now be trivial to implement, because I hooked into CIEL’s “fetching plan” mechanism. In case of a required resource that exists somewhere else in the network, the shared memory plan would simply fail, and CIEL would fallback to using HTTP methods.

4.2 Unit testing of filesystem

Large scale projects require continuous testing to catch bugs early and generally go smoothly. However, when working on a systems project, it can often be difficult to come up with a strategy for testing components that modify some internal state of the computer. It is even more challenging when the components actually modify state in a remote location, as is the case with SHMD. However, one component that could be properly unit tested was the in-memory filesystem, `shmfs`. Here I summarise how it was done.

4.2.1 Preparation

`shmfs` was designed in a way that allowed it to be built as a standalone component on regular Linux. Then, a few small tools were written: an initialiser, a loader (which performs a copy from local filesystem into `shmfs`), a storer (which does the opposite) and a link creator. Finally, test cases and a testing script (included in Appendix A) were written.

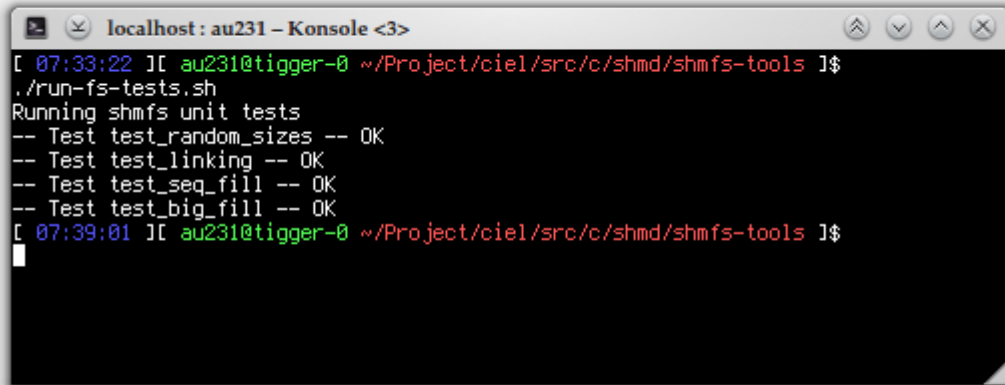
A terminal window titled 'localhost: au231 - Konsole <3>' showing the execution of a script. The prompt is 'au231@tiger-0 ~/Project/ciel/src/c/shmd/shmfs-tools'. The user runs './run-fs-tests.sh'. The output shows 'Running shmfs unit tests' followed by four tests: 'test_random_sizes', 'test_linking', 'test_seq_fill', and 'test_big_fill', all of which passed with 'OK' status. The prompt then changes to 'au231@tiger-0 ~'.

Figure 4.1: Output from the *shmfs* unit testing framework.

4.2.2 Results

The filesystem was built to contain 1024 directory entries, 1024 inodes and 1024 blocks, 1 MB each. A sample output of the test runs is displayed in Figure 4.1. The tests include creating links and checking that they work, filling the filesystem with a large file, filling with 1 block-sized files and filling with randomly sized files.

4.3 Testing machines

The rest of the evaluation work was done on two computers made available by the Systems Research Group. Here I quickly overview their configuration.

The Intel Single-Chip Cloud computer has already been presented in Section 2.2. The configuration used had 32 GB of RAM in total, thus providing 682 MB for every core. The SCC allows running under various different timings, thus for accuracy I shall state that in my configuration cores were clocked at 533 MHz, the on-chip network at 800 MHz, and DDR3 memory at 800 MHz. It runs a separate instance of the sccLinux 3.1.4 kernel on each core.

The second testing machine is a quad-processor AMD Magny-Cours system, totalling 48 cores. Each core runs at 1.9 GHz and there is 64 GB of RAM available. This is a traditional cache-coherent architecture and so runs a single instance of Linux 3.5.0.

4.4 Benchmarking of I/O scheduler

In order to investigate the effects that concurrent access has on the I/O throughput of the shared hard drive that is backing `shmfs`, I devised the following synthetic benchmark, which I ran on the AMD system.

4.4.1 Setup

Using a standalone `shmfs` and the tools mentioned in Section 4.2.1, I measured the time, t , it takes to load large files, of size, s , in parallel with different worker counts, n . Then I calculated the throughput of each run by:

$$throughput = \frac{n \times s}{t}$$

I obtained results using 128 MB and 1 GB files, running up to 48 and 10 jobs in parallel, respectively. I performed the tests with three configurations of my I/O scheduler: disabled, allowing one I/O job at a time, and allowing two jobs to run concurrently.

When doing I/O benchmarks it is important to remember that the operating system caches the hard drive contents in memory. Therefore these caches must be dropped prior to performing test runs. The script used to obtain all results is included in Appendix B.

4.4.2 Results

The hypothesis is that with multiple I/O jobs running in parallel, throughput will decrease as a result of poor seeking behaviour on the hard drive. We can see this is confirmed by my benchmark, as shown in Figure 4.2. One job runs at the full bandwidth provided by the hard drive, and adding more jobs in parallel quickly degrades performance, reducing it to nearly 50% of what is possible. An interesting feature of the graph is the increase in throughput with more than 16 parallel jobs running. It is possible that this is a result of some bottleneck in the Linux kernel, in the sense that the kernel is not managing to schedule that many I/O tasks in parallel, so those that do run experience less contention.

With the scheduler turned on, and allowing only one job to run at a time, we should see the disk sustaining its maximal throughput. Indeed Figure 4.3 shows the expected result. Tuning the scheduler to allow two concurrent jobs results in a predictable performance loss as shown in Figure 4.4. Throughput in that case is sustained at about 67 MB/s, compared to about 105 MB/s in the one job at a time scenario.

Even though limiting such contention on the hard drive maintains high throughput, we must understand the limitations of both this benchmark and the scheduling approach. First of all, the benchmark is synthetic and thus it is not representative of a real scenario. However, it is a good estimate, because the behaviour of `shmfs` is to load a file into memory

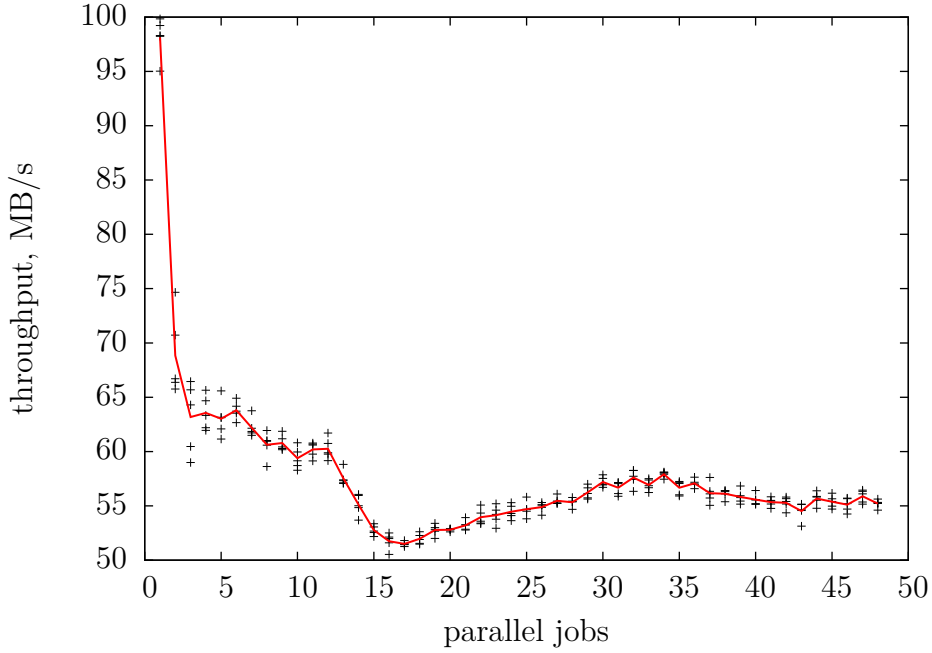


Figure 4.2: I/O read throughput without the *shmfs* scheduler.

fully, so a CIEL task which spawns multiple workers might result in many I/O jobs being run in parallel. Secondly, this is only a benchmark of I/O throughput, but overall system performance depends on more factors. The scheduling approach taken, namely blocking of I/O jobs, means that some CIEL workers will effectively waste CPU cycles if there is another task which could be run, as briefly mentioned in Section 3.6.

4.5 Further benchmarks

Originally, I was supposed to carry out extensive empirical evaluation of the different tasks that can be run on CIEL. However, as I have already mentioned, there were problems with CIEL and its example programs that set back this project quite significantly. I tried solving the bugs with the authors, but in the end there was not enough time left to perform evaluation.

The study would have included running both I/O and compute intensive tasks on the AMD and SCC machines. I would also have looked at a *timespin* example, which measures the overhead of creating a task with the CIEL system.

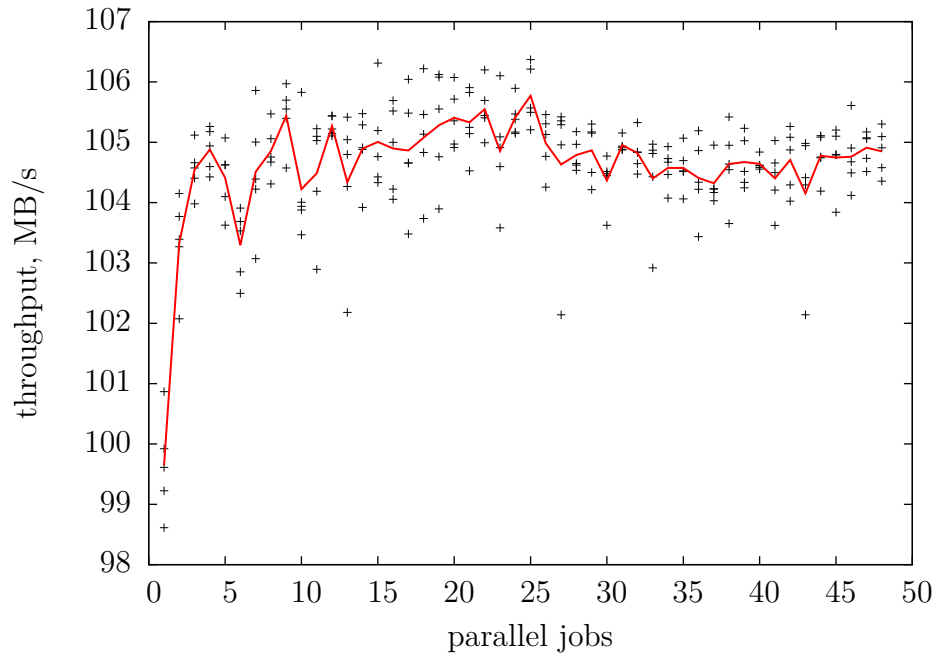


Figure 4.3: I/O read throughput with the *shmfs* scheduler, limiting to one job at a time.

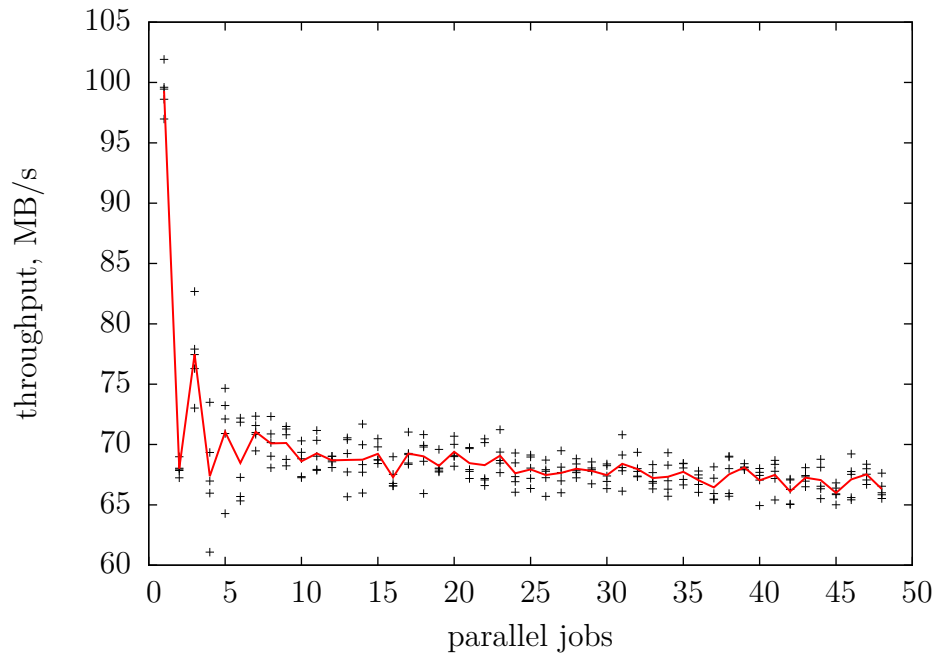


Figure 4.4: I/O read throughput with the *shmfs* scheduler, limiting to two jobs at a time.

4.6 Summary

In this chapter, I have described the evaluation of my project. I summarised what the success criteria were and related my work to them, showing that the project has, indeed, been successful.

I presented how correct functioning of the shared memory filesystem was ensured by developing unit tests, and then moved on with an empirical evaluation. The empirical study looked at concurrent I/O throughput achievable with **shmfs** and its scheduler.

Chapter 5

Conclusions

In this dissertation I have presented my work done for the Cluster-in-a-box project. I described the preparatory work and design, elaborated on the implementation, and finished off with evaluation of a shared memory controller and in-memory filesystem for the CIEL execution engine.

With a bit of hindsight I can say that if I would have to do this project again, my approach would be a bit different. Inevitably I have learned some valuable lessons which I shall talk about below. Overall, I enjoyed working on SHMD, even though it was difficult at times and there were some massive delays. There was a lot of gratification in working with the testing computers. The Single-Chip Cloud computer is an exciting development in both computer architecture and systems research, and it was a pleasure to work on it.

5.1 Achievements

I have accomplished the initial goals of the project, and in doing so gained more experience in systems programming. I believe that my `shmfs` is one of the first development attempts at a shared in-memory filesystem for the Single-Chip Cloud computer and I consider that one of the greatest achievements of the project.

5.2 Lessons learned

The hurdles in getting this project to work have shown that I underestimated the quality of research-grade code in a complex system. It would have been more prudent to propose a less diverse but more complete feature set for the project. That would have left more time to tackle the difficulties and possibly produced an even better final result.

Also, I learnt quite a lot about CIEL and the design of such distributed execution engines.

I now believe that the designers of such systems should always abstract away the details of the object storage layer, not only on the external API level, but within the system's internal code as well. It is, in principle, good software engineering practice, and also has the added benefit that multiple kinds of storage can be provided transparently. Currently CIEL does not do this abstraction consistently, and as a result the integration of my work with CIEL is not entirely seamless. It is a bit of a shame that the Firmament project was still in its infancy when this project began, as it might have been more suitable for use with SHMD and `shmfs`.

5.3 Future work

While the success criteria were met, I believe there is a lot of interesting work that can be done to improve on this project. Below are some possible improvements that I have identified:

- **Implement `shmfs` as a proper filesystem.** Currently, files in `shmfs` can be accessed only after making an explicit request through a SHMD (but this is not a hard limitation: one could, in fact, build a standalone `shmfs` request server). This proposal would look at implementing `shmfs` through FUSE or as a filesystem within the Linux kernel. That way many more applications could directly benefit from the in-memory storage provided.
- **Cache eviction and replacement.** The current implementation of `shmfs` does not support automatically moving files out into the backing hard drive once the memory space is used up. Implementing such eviction would entail keeping track of file usage, and adopting some eviction policy, such as least-recently-used. Care would have to be taken not to move out a file that is currently being read or written, and a lot of detail paid to making metadata operations (file creation, deletion, linking, etc.) atomic.
- **Many-core clusters.** This suggestion would make CIEL aware of both inter- and intra-server parallelism. It could then select the communication transport (HTTP or MPB or other) according to the situation, and also intelligently schedule tasks both within a machine and across machines.
- **Improvements to the I/O scheduler.** A simple idea is to interleave I/O jobs, while still limiting the number of them that run in parallel. This should be carefully calibrated to avoid poor seeking patterns and it might not be possible to do without losing performance. Another suggestion is to add more awareness about I/O to CIEL, so that it could make more proper decisions in scheduling.

Bibliography

- [1] Luiz Andre Barroso. Warehouse-scale computing: Entering the teenage decade. In *Proceedings of the 38th annual international symposium on Computer architecture, ISCA '11*, pages –, New York, NY, USA, 2011. ACM.
- [2] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference, DAC '07*, pages 746–749, New York, NY, USA, 2007. ACM.
- [3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7, OSDI '06*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.
- [4] Carsten Clauss, Stefan Lankes, Jacek Galowicz, and Thomas Bemmerl. iRCCE: a non-blocking communication extension to the RCCE communication library for the Intel Single-chip Cloud Computer. Version 2.0: iRCCE FLAIR. *Chair for Operating Systems, RWTH Aachen University*, 2013.
- [5] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6, OSDI'04*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
- [6] Message P Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.
- [7] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, feb. 2010.
- [8] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys '07*, pages 59–72, New York, NY, USA, 2007. ACM.

- [9] Andrew Josey. The Single UNIX Specification Version 3. *Open Group*, 2004.
- [10] Ted Kubaska. Shared Memory on Rock Creek. <http://communities.intel.com/docs/D0C-5644>. [accessed 16 October 2012].
- [11] Tim Mattson and Rob van der Wijngaart. RCCE: a small library for many-core communication. <http://communities.intel.com/docs/D0C-5628>. [accessed 08 May 2013].
- [12] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core SCC Processor: the Programmer’s View. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.
- [13] Derek G. Murray and Steven Hand. Scripting the cloud with skywriting. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud’10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.
- [14] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: a universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI’11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.
- [15] Peter Norvig. Teach yourself programming in ten years. <http://norvig.com/21-days.html>, 2001. [accessed 15 May 2013].
- [16] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, January 2010.
- [17] Christoph Rohland, Hugh Dickins, and Kosaki Motohiro. tmpfs. <https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt>. [accessed 16 May 2013].
- [18] Antony Rowstron, Dushyanth Narayanan, Austin Donnelly, Greg O’Shea, and Andrew Douglas. Nobody ever got fired for using Hadoop on a cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, HotCDP ’12, pages 2:1–2:5, New York, NY, USA, 2012. ACM.
- [19] Malte Schwarzkopf, Derek G. Murray, and Steven Hand. Condensing the cloud: running CIEL on many-core. In *First Workshop on Systems for Future Multi-Core Architectures*, pages 15–20, 2011.

- [20] Steven Smith, Anil Madhavapeddy, Christopher Snowton, Malte Schwarzkopf, Richard Mortier, Robert M Watson, and Steven Hand. The case for reconfigurable I/O channels. In *RESolve workshop at ASPLOS*, volume 12, 2012.
- [21] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 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*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

Appendix A

shmfs unit testing script

```
#!/bin/bash

(( BSIZE=1024*1024 ))
(( FSIZE=$BSIZE-1 ))
(( NBLOCKS=1024 ))
(( MAXFILE=$NBLOCKS*$BSIZE-1 ))

TESTDIR=$(mktemp -d)

reinit_fs() {
    rm -f /dev/shm/shmfs-control
    rm -f /dev/shm/shmfs-data
    ./shminit
}

make_test_file() {
    head -c$2 /dev/urandom > $1
}

get_random_number() {
    N=$(od -vAn -N4 -tu4 < /dev/urandom)
    (( N = $N % ($2 - $1) + $1 ))
    echo $N
}

test_random_sizes() {
    (( maxsize = $BSIZE * 4 - 1 ))

    for ((i = 0; i < NBLOCKS/4; i++))
    do
        make_test_file $TESTDIR/test-$i $(get_random_number 1 $maxsize)
        ./shmld $TESTDIR/test-$i
    done

    ((failed = 0))

    for ((i = 0; i < NBLOCKS/4; i++))
    do
        rm -f $TESTDIR/output
        ./shmst test-$i $TESTDIR/output
        if ! diff -q $TESTDIR/test-$i $TESTDIR/output; then
            ((failed++))
        fi
    done
}
```

```

    return $failed
}

test_seq_fill() {
    for ((i = 0; i < NBLOCKS; i++))
    do
        make_test_file $TESTDIR/test-$i $FSIZE
        ./shmld $TESTDIR/test-$i
    done

    ((failed = 0))

    for ((i = 0; i < NBLOCKS; i++))
    do
        rm -f $TESTDIR/output
        ./shmst test-$i $TESTDIR/output
        if ! diff -q $TESTDIR/test-$i $TESTDIR/output; then
            ((failed++))
        fi
    done

    return $failed
}

test_big_fill() {
    make_test_file $TESTDIR/bigtest $MAXFILE
    ./shmld $TESTDIR/bigtest
    ./shmst bigtest $TESTDIR/output
    if ! diff -q $TESTDIR/bigtest $TESTDIR/output; then
        return 1
    else
        return 0
    fi
}

test_linking() {
    make_test_file $TESTDIR/original 1024
    ./shmld $TESTDIR/original
    ./shmln original alias1
    ./shmln original alias2
    ./shmln alias1 alias3

    ((failed = 0))

    for ((i = 1; i <= 3; i++))
    do
        ./shmst alias$i $TESTDIR/alias$i
        if ! diff -q $TESTDIR/original $TESTDIR/alias$i; then
            ((failed++))
        fi
    done

    return $failed
}

run_test() {
    reinit_fs
    rm -f $TESTDIR/*
    $1
}

TESTS="test_random_sizes test_linking test_seq_fill test_big_fill"

echo "Running shmfs unit tests"

```



```
for t in $TESTS
do
    echo -n "-- Test $t"
    if run_test $t; then
        echo " -- OK"
    else
        echo " -- FAIL"
    fi
done

rm -r $TESTDIR
```


Appendix B

I/O benchmarking script

```
#!/bin/bash

TDIR=$1
DIR=$(basename $1)
WORKERS=$2
RUNS=$3
MAXWRITES=$4

RESDIR=results/$DIR-mw$MAXWRITES-$(date +%s)
mkdir -p $RESDIR

run_test() {
    rm -f /dev/shm/shmfs-control /dev/shm/shmfs-data
    drop_caches
    shminit

    time (ls $TDIR/file-* | head -n$2 | xargs -I'{}' -n1 -P$2 shmls '{} ' $MAXWRITES) &> $1
}

echo "I/O scheduling benchmark"
echo

for ((i = 1; i <= $WORKERS; i++))
do
    echo "---- Testing $i workers ----"
    echo
    for ((j = 0; j < $RUNS; j++))
    do
        echo -n "date' -- Doing run $j"
        run_test $RESDIR/x${i}_run-${j} $i
        echo " -- done"
    done
    echo
done
```


Appendix C

Test generation for I/O benchmark

```
#!/bin/bash

SIZE=$1
NUMBER=$2
DIR=$3

mkdir $DIR

bashstring="head -c$SIZE /dev/urandom > $DIR/file-{"
seq 1 $NUMBER | xargs -n1 -I{} -P$NUMBER bash -c "$bashstring"
```


Appendix D

Project proposal

Antanas Uršulis
Queens' College
au231

Part II Computer Science Project Proposal

Cluster-in-a-box: task-parallel computing on many-core machines

May 17, 2013

Project Originator: Malte Schwarzkopf

Resources Required: See attached Resource Declaration

Project Supervisor: Malte Schwarzkopf

Director of Studies: Dr. Robin D. H. Walker

Overseers: Dr. Andrew C. Rice and Dr. Timothy G. Griffin

Introduction and Description of the Work

Recent years have witnessed the emergence of various task-parallel execution engines, such as Google's MapReduce [5], Microsoft's Dryad [8] or Cambridge's own CIEL [14]. They provide programmers with easy to use abstractions for implementing distributed algorithms, and transparently manage the often gory details of distributing code and data across the network and dealing with faults. These systems were designed when typical computers had only a few (1–4) processor cores, and the algorithms would execute on a large number of computers within a datacentre network. However, nowadays, machines are available with tens or hundreds of cores, and non-cache-coherent architectures, such as Intel's Single-Chip Cloud Computer [7], could potentially scale to thousands of cores with a suitable manufacturing process. Such architectures allow us to perform task-parallel computation on a single machine, but current execution engines are not optimized for this environment. A number of issues with running CIEL on many-core machines have been identified [19], and my project aims to address them.

The engine has to provide an object store, which tracks object metadata (where does each object reside?). Worker instances keep the objects on their local disks, but the metadata has to be managed by a coordinator (master) instance. Many accesses to the metadata or retrieving task arguments (which are also served by the object store in the master) will contend the master instance. This kind of centralised metadata service might be necessary in a distributed system, but in our single machine scenario, the store can be shared between workers.

Furthermore, instances of the engine usually communicate via TCP/IP (or, in the case of CIEL, HTTP over TCP/IP), which incurs a lot of avoidable overhead, as the many-core architectures typically have more efficient internal interfaces that provide reliable messaging. One optimisation approach would be to implement architecture-independent communications over shared memory. Then, it is possible to optimise further by using platform specific features, like the message passing buffers in Intel's SCC.

When tasks become fine-grained (consisting of maybe only a few instructions), the master instance is contended with updates to its data structures (for example, the task graph). If the updates are to independent regions, then allowing concurrent updates could improve performance. Securing access to the structures could be done by fine-grained locking.

The object store is usually backed by hard disks, thus the most efficient access pattern is long sequential reads/writes. The engine should incorporate a scheduler that does not cause I/O contention, by limiting the amount of concurrent I/O-bound tasks to a number proportional to the number of disks in the machine.

Another inefficiency is that intermediate results of computations are stored on disk, while on a single machine they could be stored and passed between tasks in shared memory.

Starting Point

This project will build on the codebases of the following two task-parallel distributed execution engines, both developed by the Systems Research Group in the University of Cambridge Computer Laboratory:

- CIEL <http://www.cl.cam.ac.uk/research/srg/netos/ciel/>
- Firmament <http://www.cl.cam.ac.uk/~ms705/research/firmament/>

I have taken the following Computer Science Tripos courses which are relevant to the project: Part IA Operating Systems, Part IB Computer Design, Part IB Concurrent and Distributed Systems, Part IB Computer Networking, Part IB Programming in C and C++.

Having done an internship at the SRG over the summer of 2012, I will use the experience I gained on building systems and using C/C++.

Substance and Structure of the Project

The goal of the project is to optimise the CIEL execution engine to run on a many-core computer which does not have cache-coherency. A number of optimisations have been outlined in the introduction, I aim to implement them one at a time, marking version control system tags for each feature completed. This way I can evaluate the project at different stages and also have an option to return to them for further evaluation.

Starting with the CIEL code, I shall do the following. Points 1 and 2 relate to optimising the usage of objects (data plane of the system). Points 3 and 4 relate to optimising the control plane of the system. Point 6 is a design consideration for the whole system that will allow further optimisation. Finally, point 5 builds on the points 1–4 and is dependent on them being implemented.

1. **Share object storage.** Instead of retrieving and saving objects via CIEL's HTTP server, worker instances can instead share the state of the object store and access it directly.
2. **Do not write back intermediate results.** Currently CIEL stores every produced object on disk, and tasks read input from disk (or in the distributed cluster case, fetch over the network). This part of the project would implement storing intermediate objects in memory. They will then be passed between tasks by sharing memory pages. For Intel's Single-Chip Cloud, this part can be tricky as the current SCC Linux implementation allows only 64 MB of shared memory accessible from all cores. Thus the data could either be transferred in several passes or, alternatively, the available shared memory could be extended [10].

3. **Replace TCP/IP control message flow.** CIEL was written for a distributed environment and its instances exchange control messages through TCP/IP. There are more efficient means of communication available on a single machine. This first optimisation step would replace the existing messaging mechanism with a shared-memory ring buffer. This implementation ought to be platform-independent, it will likely build on the shared memory primitives developed for the previous task.
4. **Implement platform-specific communication.** The Single-Chip Cloud has fast on-die message passing buffers which enable communication between cores. This stage of the project would replace the previously developed ring buffer communication mechanism for the SCC. If time permits and resources are available, I will also look at platform-specific features of many-core ARM machines (this can be regarded as an extension).
5. **Limit I/O contention.** Because the most efficient access pattern to spinning platter hard drives is long sequential reads or writes, concurrently scheduling two or more tasks which perform lots of I/O can greatly reduce performance by causing the disk to seek back and forth many times. For this stage of the project, a scheduling algorithm would be devised and implemented, so that it limits the number of tasks performing I/O based on the number of hard drives available in the machine. The scheduler would interact with the object store to know that I/O is requested, and also estimate the time it would take based on the object size. The scheduler is dependent on the previous points being implemented, because its design will take into account characteristics such as IPC overhead, details of how objects are managed.
6. **Investigate locking of CIEL's structures.** Fine-grained locking of CIEL's dynamic task graph and other structures could increase concurrency in the coordinating instance, thus allowing more workers to make progress. This part of the project would look into making the locking of those structures fine-grained and then evaluating whether this gives a speedup or if the locking overhead slows down the computation. This part comes last as it modifies structures which might have been reimplemented in previous parts. Throughout the project, it will be good practice to design structures such that they are suitable for fine-grained locking.
7. **Evaluate.** The project can be evaluated by using it to perform large scale computations, such as the ones used in the original CIEL paper [14]. These include grep, k -means clustering and binomial option pricing, among others. Results will be produced for each incremental improvement (and different combinations of optimisations, if possible). Then the total time taken, the parallel speedup, and other characteristics (if reasonable) will be compared against the non-optimised version. Another good indicator of the success of the optimisations is relative overhead [19].

Possible Extensions

1. Implement support for mixed clusters:
 1. Tasks can be scheduled on different cores within a single machine, and also sent to a different machine across the network.
 2. Implement an intelligent scheduler for the mixed cluster. It should be able to estimate: the cost of sending a task to another machine, the cost of creating I/O contention on the current machine. Using these estimates it can then make appropriate decisions where to schedule a task.
2. Measure power consumption and throughput when running computations. This will allow to quantify how much work is done per kilowatt and the results can be compared to those of conventional clusters.

Success Criteria

For this project to be considered a success, I must have accomplished the following:

1. Have modified a task-parallel data flow execution engine to share its object store between instances and transfer intermediate results via shared memory.
2. Have modified the engine to use shared memory instead of network-based communication.
3. Implemented at least one architecture-specific optimisation.
4. Implemented a scheduler that limits the number of tasks performing I/O concurrently.
5. Investigated use of fine-grained locking to improve concurrency.
6. Compared the performance of various large-scale computations on the optimised and network-based variants of the engine.

Implementing extensions detailed in the previous section would make the project more successful, but they are not essential.

Timetable and Milestones

About 30 weeks are available after the proposal submission to complete the project and write a dissertation. Thus I am planning to split this time into 3-week parts according to the timetable below:

Part 0: October 1st – October 19th

Sort out project details and write proposal. Begin doing research into the topic.

Milestone: Write project proposal.

Part 1: October 20th – November 9th

Research. I should investigate in detail the target architectures, existing codebases, libraries or interfaces used to manipulate platform-specific features. Learn about the use of shared memory pages and ring buffers. Learn about scheduling techniques beyond what was taught in the IA Operating Systems course. Begin setting up the coding environment and try out evaluation workloads.

Part 2: November 10th – November 30th

Begin implementation phase. Implement the optimisations related to the object store.

Milestone: Have at least implemented shared access to the store.

Part 3: December 1st – December 21st

Finish off object store optimisations and replace TCP/IP control messages with the shared memory implementation. Then implement architecture-specific communication.

Milestone: Instances communicate via shared memory/special interfaces, pass intermediate results in shared memory.

Part 4: December 22nd – January 11th

Implement the I/O limiting scheduler and work on extensions.

Part 5: January 12th – February 1st

Finish working on features. Prepare a progress report and presentation.

Milestone: Hand in progress report and have a prepared presentation. Be feature complete by the end of timeslot.

Part 6: February 2nd – February 22nd

Buffer time, debugging of code and evaluation. Begin writing dissertation: think of the structure and produce introductory text.

Part 7: February 23rd – March 15th

Write the core of the dissertation. Produce figures from evaluation results, any other relevant parts.

Milestone: Have a draft of the dissertation.

Part 8: March 16th – April 5th

Finish the dissertation. Comments from supervisor and other issues will be addressed.

Part 9: April 6th – April 26th

Buffer time, revision for exams.

Part 10: April 27th – May 17th

Buffer time, revision for exams.

Milestone: Dissertation handed in.