A distributed execution engine supporting data-dependent control flow

Derek Gordon Murray

University of Cambridge
Computer Laboratory
King’s College

July 2011

This dissertation is submitted for the degree of Doctor of Philosophy
Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

This dissertation does not exceed the regulation length of 60,000 words, including tables and footnotes.
A distributed execution engine supporting data-dependent control flow

Derek G. Murray

Summary

In computer science, data-dependent control flow is the fundamental concept that enables a machine to change its behaviour on the basis of intermediate results. This ability increases the computational power of a machine, because it enables the machine to execute iterative or recursive algorithms. In such algorithms, the amount of work is unbounded a priori, and must be determined by evaluating a fixpoint condition on successive intermediate results. For example, in the von Neumann architecture—upon which almost all modern computers are based—these algorithms can be programmed using a conditional branch instruction.

A distributed execution engine is a system that runs on a network of computers, and provides the illusion of a single, reliable machine that provides a large aggregate amount of computational and I/O performance. Although each individual computer in these systems is a von Neumann machine capable of data-dependent control flow, the effective computational power of a distributed execution engine is determined by the expressiveness of the execution model that describes distributed computations.

In this dissertation, I present a new execution model for distributed execution engines that supports data-dependent control flow. The model is based on dynamic task graphs, in which each vertex is a sequential computation that may decide, on the basis of its input, to spawn additional computation and hence rewrite the graph. I have developed a prototype system that executes dynamic task graphs, and discuss details of its design and implementation, including the fault tolerance mechanisms that maintain reliability throughout dynamic task graph execution. Dynamic task graphs support a variety of programming models, and I introduce a model based on multiple distributed threads of execution that synchronise deterministically using futures and continuations. To demonstrate the practicality of dynamic task graphs, I have evaluated its performance on several microbenchmarks and realistic applications, and it achieves performance that is similar to or better than an existing, less-powerful execution engine.
Acknowledgements

Foremost, I would like to thank my supervisor, Steve Hand, for his help and encouragement over the past four years. Through many hours of meetings, and his comments on countless drafts of this document, Steve’s feedback has been vital in helping me to shape my thesis.

The system at the centre of this dissertation, CIEL, has grown from one student’s thesis project to become a thriving collaborative effort. The other members of the CIEL team are Malte Schwarzkopf, Chris Smowton, Anil Madhavapeddy and Steven Smith, and I am indebted to all of them for their contribution to the project’s success. CIEL also spawned a Part II undergraduate project, and I thank Seb Hollington for being such an enthusiastic supervisee.

In addition to the team members, several current and former colleagues in the Computer Laboratory have commented on drafts of this dissertation. I am grateful to Jon Crowcroft, Stephen Kell, Amitabha Roy, Eiko Yoneki and Ross McIlroy for their comments and suggestions, which have greatly improved the clarity of my writing. I would also like to thank Andy Warfield of the University of British Columbia for his frequent invitations to give talks in Vancouver; as well as being enjoyable, the feedback that I have received has been useful in developing the ideas that I present in this dissertation.

The genesis of CIEL can be traced to the summer internship that I spent at Microsoft Research in Silicon Valley. I thank Michael Isard and Yuan Yu for giving me the opportunity to work on the Dryad project. Working with Dryad gave me useful experience in using a real-world distributed system, and helped me to realise that there was an opportunity to develop a more powerful system.
# Contents

1 Introduction .................................................. 11
   1.1 Contributions .............................................. 12
   1.2 Outline ..................................................... 13
   1.3 Related publications ...................................... 14

2 Background and related work ................................... 15
   2.1 Scales of parallelism ...................................... 17
   2.2 Parallel programming models .............................. 25
   2.3 Coordinating distributed tasks ............................ 36
   2.4 Summary .................................................... 46

3 Dynamic task graphs ............................................ 47
   3.1 Definitions .................................................. 48
   3.2 Executing a dynamic task graph ........................... 55
   3.3 Relation to other execution models ...................... 59
   3.4 Summary .................................................... 70

4 A universal execution engine .................................. 72
   4.1 Distributed coordination .................................. 73
   4.2 A simple distributed store ................................ 80
   4.3 Scheduling ................................................... 87
   4.4 Fault tolerance .............................................. 95
   4.5 Summary .................................................... 99
5 Parallel programming models

5.1 Implementing existing models .................................. 100
5.2 First-class executors ............................................. 105
5.3 Distributed thread programming model ......................... 109
5.4 Summary .......................................................... 125

6 Evaluation

6.1 Experimental configuration ....................................... 129
6.2 Task handling ....................................................... 129
6.3 MapReduce .......................................................... 134
6.4 Iterative k-means .................................................... 138
6.5 Fault tolerance ....................................................... 141
6.6 Streaming ............................................................ 144
6.7 Summary ............................................................ 147

7 Conclusions and future work

7.1 Extending dynamic task graphs ................................. 150
7.2 Alternative system architectures ............................... 151
7.3 Separating policy from mechanism ............................ 152
7.4 Summary ............................................................ 153
## List of Figures

2.1 An outline of the topics that have influenced dynamic task graphs .......................... 16
2.2 Models of MIMD parallelism ................................................................................. 19
2.3 A message-passing version of Conway’s Game of Life ......................................... 28
2.4 A simple data-flow graph ..................................................................................... 31
2.5 Basic task farm architecture ................................................................................. 36
2.6 Data-flow graphs for various models of task parallelism ...................................... 37
2.7 Example of acyclic data-flow in `make` ................................................................. 41
3.1 Symbols representing a concrete object and a future object .................................... 48
3.2 Illustration of a store with concrete and future objects .......................................... 49
3.3 Illustration of a task with dependencies and expected outputs ................................ 50
3.4 Illustration of task output delegation ..................................................................... 54
3.5 Lazy evaluation algorithm for evaluating an object in a dynamic task graph .......... 58
3.6 Dynamic task graphs for recursively calculating the $n^{th}$ Fibonacci number ...... 60
3.7 Dynamic task graph for performing a MapReduce computation ............................. 62
3.8 Dynamic task graph for the first two supersteps of a Pregel computation .......... 65
3.9 Implementation of mutable state in a dynamic task graph ..................................... 67
3.10 Dynamic task graph for two iterations of a while loop .......................................... 68
4.1 The reference lifecycle ......................................................................................... 72
4.2 A CIEL cluster has one or more clients, a single master and many workers .......... 73
4.3 Example job descriptor for a $\pi$-estimation job .................................................... 77
4.4 Pseudocode for the CIEL master and worker ....................................................... 78
4.5 Representation of a Java example task and its code dependency ............................ 79
4.6 Expected load distribution using two different replica placement strategies .......... 84
4.7 Proportion of data-local tasks as the cluster size is varied ... 89
4.8 Distribution of input object replication factors as an iterative job executes ... 90
4.9 Non-local tasks in the first 20 iterations of $k$-means clustering ... 93
4.10 Execution time for the first 20 iterations of $k$-means clustering ... 94
4.11 Mechanisms for providing master fault tolerance. ... 98
5.1 Pseudocode for a MapReduce map task ... 104
5.2 Rules for computing the result of the executor function for a first-class task ... 105
5.3 Dynamic task graph for a distributed thread, comprising three tasks ... 110
5.4 A continuation in a distributed thread ... 111
5.5 Dynamic task graphs for Skywriting tasks ... 115
5.6 Dynamic task graph for a Skywriting script that blocks on a child task ... 115
6.1 A synthetic sequential workload ... 131
6.2 A synthetic parallel workload ... 132
6.3 A synthetic iterative workload ... 133
6.4 Comparison of Hadoop and CIEL absolute performance for $\pi$ estimation ... 136
6.5 Comparison of Hadoop and CIEL parallel speedup for $\pi$ estimation ... 136
6.6 Comparison of Hadoop and CIEL absolute performance for Grep ... 137
6.7 Comparison of Hadoop and CIEL execution time for $k$-means clustering ... 139
6.8 Comparison of MPI and optimised CIEL execution time of $k$-means clustering ... 139
6.9 Distribution of master overhead with and without fault tolerance ... 141
6.10 The overhead of master fault tolerance under randomly-induced failures ... 143
6.11 Streaming throughput in TCP and HTTP modes ... 145
6.12 Illustration of the data dependencies in a BOPM calculation ... 146
6.13 Speedup of BOPM for various problem sizes ... 147
7.1 Possible representation for non-deterministic select in a dynamic task graph ... 152
List of Tables

3.1 Definition of a dynamic task graph for calculating the $n^{th}$ Fibonacci number . . 60
3.2 Definition of a dynamic task graph for performing a MapReduce computation . 61
3.3 Definition of a dynamic task graph for performing a Pregel computation . . . . 64
3.4 Definition of a dynamic task graph for executing a while program . . . . . . 68
4.1 Rules for combining an existing reference with an incoming reference . . . . . 76
4.2 Rules for combining an existing reference with an incoming stream reference . 85
4.3 Rules for combining an existing reference with an incoming sweetheart reference 91
4.4 Rules for combining an existing reference with an incoming tombstone reference 97
5.1 Rules for combining a fixed reference with an incoming tombstone reference . 124
Listings

5.1 Simple Java tasks must implement the `Task` interface.

5.2 First-class Java tasks must implement the `FirstClassJavaTask` interface.

5.3 Skywriting script that spawns a single task and blocks on the result.

5.4 Skywriting script to perform `detex thesis.tex | wc -w`.

5.5 Skywriting script for computing the 10th Fibonacci number.

5.6 Example of the control flow when using the `reset` and `shift` operations.

5.7 The Scala thread entry point uses `reset` to delimit the continuation.

5.8 The Scala `Future[T].get` method uses `shift` to capture the continuation.

5.9 Example of Scala threads, showing static types.

5.10 Example of Scala threads, using type inference.

5.11 Example of Scala generator tasks.

5.12 Scala program for computing the 10th Fibonacci number.

5.13 Checkpoint-based C program for computing the 10th Fibonacci number.
Chapter 1

Introduction

A distributed execution engine is a software system that runs on a cluster of networked computers, and presents the illusion of a single, reliable machine. Distributed execution engines are attractive because they shield developers from the challenging aspects of distributed and parallel computing, such as synchronisation, scheduling, data transfer and dealing with failures. Instead, the developer writes programs against an execution model that exposes high-level operations for performing parallel computations.

In this dissertation, I focus on the expressive power of these execution models. The most primitive execution model allows the developer to specify a finite set of independent tasks, which can then be executed in parallel [BBK+87, And04]. While some computations can be decomposed into independent tasks, it is often necessary impose some order on the tasks. Dean and Ghemawat’s MapReduce system [DG04] is an execution engine that schedules tasks in two discrete phases, and Isard et al.’s Dryad [IBY+07] extends this to an arbitrary directed acyclic graph (DAG). However, because the set of tasks in these execution models is finite, the overall execution time is bounded, which means they can only support functions that are primitive recursive [MR67]. Iterative and general recursive functions require additional coordination logic that performs data-dependent control flow. Intuitively, this logic evaluates a predicate on the intermediate data and, depending on the result, either schedules more computation or terminates.

In the research for this dissertation, I have developed a system called CIEL, which is a distributed execution engine that supports data-dependent control flow within a single computation. I use this system to argue the following thesis:

Data-dependent control flow can be supported in a distributed execution engine by adding the facility for a task to spawn further tasks. The resulting execution model is capable of representing all functions that are computable by a Universal Turing Machine. A system implementing this model can achieve the same non-functional properties—such as fault tolerance and performance—as an existing less computationally-powerful execution engine.
1.1 Contributions

In this dissertation, I make three principal contributions:

1. My first contribution is the **dynamic task graph** execution model for parallel computation. Dynamic task graphs extend the static DAG execution model to form a labelled transition system, in which each state is a static task graph. The transition relation captures the effect of executing a task: it transforms the current state into a new task graph that may contain additional tasks. I formalise the semantics of dynamic task graphs, and show that the transition relation is **commutative**, **idempotent** and **monotonic**. These properties imply that dynamic task graphs are suitable for representing parallel computation in an unreliable distributed system. Furthermore, I show that dynamic task graphs are Turing-complete, even when the amount of work in a single task is bounded, by devising a reduction from a Turing-complete model of computation (while programs [BJ66]) to dynamic task graphs.

2. My second contribution is the **CIEL system architecture**, which is a concrete instantiation of the dynamic task graph execution model. CIEL is designed to support distributed execution in a networked cluster of commodity servers. To support this environment, it is necessary to tolerate faults in the underlying machines, and limit the amount of data that is transferred across the network. I have devised techniques that enable a CIEL computation to continue making progress in the presence of faults, and use the structure of an iterative computation to obtain an efficient schedule.

3. My third contribution is the **distributed thread programming model**. A CIEL computation can be specified by writing one or more programs in a Turing-complete language. These programs can be automatically transformed into chains of tasks (**threads**), which facilitates deterministic synchronisation between the programs. To implement distributed threads, I have created a programming language called Skywriting, which is a hybrid functional-imperative scripting language that supports dynamic task creation. Furthermore, the techniques for decomposing a distributed thread into tasks can be applied to existing languages, and I describe implementations that use Scala delimited continuations [RMO09] and an OS-level checkpointing framework [DHR02].

The design and initial implementation of all models described in this section are the result of my own work. However, some of the components that I will describe later in this dissertation have been implemented or extended by other students in the Computer Laboratory. In particular, Christopher Smowton extended the streaming implementation to support direct TCP connections (§4.2.3) and collaborated in the development of the executor interface (§5.2.2). Sebastian Hollington implemented the support of OS-level checkpointing (§5.3.3). In addition, Malte Schwarzkopf, Anil Madhavapeddy, Steven Smith and Steven Hand have co-authored various papers about CIEL [MH10, MSS+11, SMH11, MH11].
1.2 Outline

This dissertation is structured as follows:

Chapter 2 traces the development of parallel programming in order to identify the concepts that have influenced the development of distributed execution engines. In particular, I focus on systems that have been developed for parallel execution on loosely-coupled clusters, and show how the class of problems that these systems can solve efficiently has expanded in recent years.

Chapter 3 formalises the dynamic task graph execution model that is the subject of my thesis. I define the primitive entities in the model (tasks and data objects), and how they can be composed to build and evaluate a dynamic task graph. I also show how existing execution models can be expressed as dynamic task graphs, and use this approach to show that dynamic task graphs are Turing-complete.

Chapter 4 introduces CIEL by explaining how it executes a dynamic task graph. The key concept in this chapter is the reference lifecycle: references are used to represent each data object in a CIEL cluster throughout a computation, including before it is created, when it is stored on one or more machines, and after it is lost due to machine failure. I show how CIEL uses references to support a variety of non-functional features, including data replication, streaming, fault tolerance and data-local scheduling.

Chapter 5 discusses various programming models that can be built on top of dynamic task graphs, and hence used to implement CIEL computations. I consider three models in this chapter: (i) a simple model that can be used to implement static (bounded) computations, (ii) a tail-recursive model that enables iteration by programming in continuation-passing style and (iii) the distributed thread model, which transparently uses continuations to implement deterministic synchronisation.

Chapter 6 evaluates the performance of CIEL when running several different synthetic benchmarks and realistic applications. I compare the performance to an implementation of MapReduce (Hadoop [Had]), and a lower-level message passing library (MPI [Mes94]). This chapter demonstrates that CIEL can achieve performance that is competitive both with less expressive systems, and with lower level approaches that do not provide fault tolerance.

Chapter 7 concludes this dissertation, and outlines directions for future work. In particular, I consider the following question: what programs cannot—efficiently or at all—be represented as dynamic task graphs?
1.3 Related publications

Some of the work presented in this dissertation has previously been published in various venues. The relevant publications are:


Chapter 2

Background and related work

Parallel computing can be divided into two distinct but related problems:

**Computation** This problem involves specifying the operations that a processor should execute in order to produce a correct result.

**Coordination** This problem involves specifying the temporal relation between computations executed by one or more processors.

In this dissertation, my primary contributions are a new execution model (dynamic task graphs) and programming model (distributed threads) for parallel coordination. Therefore, in this chapter, I will survey the existing hardware, software and programming models that currently enable parallel computation.

I will begin this chapter by considering the different scales at which parallelism can be achieved, and the mechanisms that support this parallelism (Section 2.1). I will then discuss how various programming models can be used to express the coordination between parallel computations (Section 2.2).

In Section 2.3, I focus on task parallelism, which has become a dominant paradigm for large scale distributed programming, because dividing a computation into independent tasks simplifies the implementation of fault tolerance and scheduling. As I introduced in Chapter 1, distributed execution engines are systems that can execute task-parallel computations reliably, and I survey the existing systems while paying particular attention to the expressivity of their execution model.

**How to read this chapter** This chapter covers a broad range of topics in parallel computing, and some topics have greater relevance to my thesis than others. Figure 2.1 (overleaf) is an attempt to extract the most important background topics and systems that have influenced the development of my research. An edge in the figure indicates where one topic has influenced the development of another, and the labels correspond to the sections in which I discuss these topics.
Figure 2.1: An outline of the topics in this chapter that have influenced the development of dynamic task graphs. An edge from topic $x$ to topic $y$ indicates that $x$ has influenced the development of $y$. 
2.1 Scales of parallelism

Parallelism can be achieved at many different scales. At the microarchitectural scale, computers are inherently parallel, since the individual logic gates in a microprocessor can operate independently on the bits in a single word. However, as the scale of operation increases, more hardware and software is required to maintain the correctness of parallel operation, which limits the granularity of computation that can be coordinated efficiently. This section surveys the different scales of parallelism in increasing order of scale (and decreasing order of coupling), ranging from a single computer (§2.1.1) and parallel computers (§2.1.2), to local networks (§2.1.3) and wide-area networks (§2.1.4).

2.1.1 Individual computer

The earliest computing machines were capable of parallel operation. Babbage’s Difference Engine was a mechanical calculator for calculating the values of polynomial functions by the method of finite differences, and it used parallelism within the addition mechanism to achieve higher utilisation [Bro83]. Hollerith’s original tabulator was a special-purpose electromechanical device for counting census data that was stored on punched cards, and it provided multiple counters for each category [Hol89]. The original (electronic) ENIAC computer originally had to be physically reconfigured in order to re-program it, but the flexibility allowed multiple “accumulators” to perform arithmetic operations in parallel [GG96]. These early machines all coordinated parallel execution using deliberate hardware design, which could not be reconfigured by the software running on that hardware.

Though early von Neumann-architecture machines provided a sequential execution model, parallelism still arose from interactions with input/output devices operating at much slower speeds than the CPU [Gil58]. Since I/O devices have finite buffers, it is necessary to coordinate between the program (or operating system) running on the CPU and the device to ensure that those buffers are not overrun. The two approaches to this coordination problem are polling and interrupts [PH94, pp. 566–570]. In a polling configuration, the CPU repeatedly reads from an I/O register or memory location that stores the device status, until it observes that the status has changed, which wastes resources if the CPU has other work to do. By contrast, in an interrupt-driven system, the device sends an asynchronous signal to the CPU, which causes it to jump to the appropriate interrupt service routine, which contains code to handle the interrupt. The difficulty of keeping a computer busy while an individual program is blocked on I/O led to the development of time-sharing—the basis of modern operating systems—in which multiple user programs share a single processor [CMDD62]. Although, on a single processor, this

\[1\] In later years, the ENIAC was rewired for stored program operation with a von Neumann architecture. As a result, it could only be programmed sequentially, and its peak operating performance was degraded by a factor of six [Neu06].
is technically *concurrency* and not parallelism, the programming mechanisms for coordinating programming concurrent processes are similar to those used in parallel programming (§2.2).

Many algorithms involve repeating the same computation across a large amount of data. *Single Instruction Multiple Data* (SIMD) is the most coupled form of this, whereby a single machine instruction is applied simultaneously to several memory locations [Fly72]. The original SIMD processors used multiple simple arithmetic and logic units to perform the same instruction in parallel; however, as the relative cost of instruction decode has shrunk compared to accessing memory, most SIMD implementations have focused on processing vectors in a data pipeline. For example, modern x86 processors include Streaming SIMD Extensions (SSE) technology, which enables a single instruction to perform parallel computation on a vector of up to 128 bits [Int11, §5.4–11]. The principal application of these instructions is multimedia processing, and it follows that Graphical Processing Units (GPUs) also make extensive use of SIMD processing: a modern GPU may contain several 32-way SIMD processing units in order to achieve high throughput [LNOM08], and programming tools such as CUDA [NBGS08] and OpenCL [SGS10] expose this facility to application programmers. However, SIMD vector processing is only suited to data-intensive applications in which the same operation is applied uniformly across a large region of memory.

To address the perceived shortcomings of SIMD, *superscalar* architectures were proposed. A superscalar processor is one capable of *instruction-level parallelism* (ILP): it executes a single stream of instructions, but it can dispatch different instructions to multiple execution units simultaneously [AC87]. To achieve parallelism, it is necessary to identify dependencies between instructions, schedule them in order of those dependencies, and dispatch the instructions in parallel. The compiler and the processor may cooperate in identifying dependencies and ordering instructions. Although the parallelism that ILP can extract is transparent to the developer, the benefits are limited. Wall studied the simulated execution of various benchmarks on an ensemble of realistic and idealised hardware configurations, and found that—under optimistic assumptions—the median speedup achievable is approximately $5 \times$ [Wal91].

Instruction-level parallelism can only extract what parallelism exists in a single stream of instructions; however, in 1979, Kaminsky and Davidson proposed that it would be more cost-effective to share the processor’s control and functional units between multiple instruction streams [KD79]. Later, Tullsen *et al.* developed *simultaneous multi-threading* (SMT), which is an alternative superscalar design that allows instructions from multiple threads (instruction streams) to be dispatched at once [TEL95]. The principle behind SMT is that maintaining multiple instruction streams in the CPU will lead to higher throughput, because when one stream experiences a pipeline stall, the CPU is able to issue instructions from another stream. In simulation, Tullsen *et al.* improved benchmark performance by approximately $4 \times$ compared to a single-threaded superscalar processor with the same hardware capabilities. Intel implemented a version of SMT (known as “Hyper-Threading”) for its Xeon processor family, and later extended it to other architectures [MBH+02]. However, as Tullsen *et al.* anticipated—and Bulpin later demonstrated [Bul04]—performance interactions between concurrent threads may lead to
Figure 2.2: Models of MIMD parallelism. (a) In a multiprocessor, many processors share the same memory. (b) In a multicomputer, each processor has private memory, and the processors communicate using special-purpose messaging hardware.

vastly different performance both for individual threads and in terms of overall throughput. If a system is to achieve highly-parallel execution on a tightly-coupled platform, it must be optimised at a low level. Field programmable gate arrays (FPGAs) offer a platform for building custom digital logic that may easily be reconfigured [CDF+86]. Since—like any combinatorial circuit—an FPGA may be configured to use a large number gates independently, FPGAs are capable of high degrees of parallelism. However, the traditional programming model, based on hardware description languages, is conceptually similar to reconfiguring patch cables on the ENIAC: considerable sophistication is required to achieve good performance. Attempts have been made to provide a high-level programming model for FPGA synthesis, but it is currently not possible to compile an arbitrary high-level program to a hardware description. For example, Greaves and Singh’s Kiwi system translates C5 programs into FPGA circuits, but prohibits unbounded (data-dependent) recursion and dynamic object allocation [GS08]. The Lime programming language is a dialect of Java that can be compiled to run on a heterogeneous system including FPGAs, but the portion of code that can run on an FPGA is limited to operations on bounded arrays with a fixed number of iterations [ABCR10]. In contrast to these general-purpose systems, Bluespec uses a dialect of Haskell as a hardware description language, which allows hardware developers to use the power of Haskell’s type system, and can facilitate provably-correct hardware designs [HHJW07].

2.1.2 Multiprocessors and multicomputers

The next scale to consider is a single computer that can execute multiple “programs” simultaneously. The definition of a “program” is here left deliberately vague: depending on the system, it may include the operating system notions of a thread or process. Multiple Instruction Multiple Data (MIMD) is a model of parallelism in which multiple instruction streams can execute, in
parallel, on multiple (possibly-overlapping) regions of memory [Fly72]. Bell uses the terms *multiprocessor* to denote a MIMD machine with multiple processors accessing a shared memory, and *multicomputer* to denote a MIMD machine in which each process has private memory and a separate communication mechanism for coordination (Figure 2.2) [Bel89].

The first multiprocessors were developed in the late 1950’s. Preliminary designs for the IBM 7030 Stretch [Dun57] and UNIVAC-LARC [Eck57] were first presented in December 1957, and each had an architecture in which two or more processors could access the shared memory; however, the multiprocessing component of each machine was removed before the first systems were delivered [Ens77]. In 1958, The Compagnie des Machines Bull presented the first Gamma 60 multiprocessor, which had an instruction code that allowed the creation and synchronisation of multiple instruction streams, and an architecture with multiple processing units [Dre58]. In the Gamma 60, bus access to the shared memory was arbitrated by a central “data distributor”, which serviced requests from the processing units in a round-robin order. Twenty Gamma 60 computers were delivered [Bat72], the first in 1960 [Fla88, p. 153]. However, the inherent assumption in the Gamma 60 design was that instruction execution is slow compared to memory access, and practical use of the Gamma 60 led to poor overall performance due to switching overhead [Ryl61]. The Burroughs D825, also first delivered in 1960, supported four computing modules that could simultaneously access up to sixteen memory modules, using a crossbar switch [AHSW62], and Enslow considered it to be the “first true multiprocessor” (original emphasis) [Ens77].

Coordination in a multiprocessor is largely achieved by writing to and reading from shared memory locations. However, as processor clock speeds have increased relative to memory access speeds, the importance of *caches* has grown. A cache is a relatively small and fast region of memory that temporarily stores a subset of main memory, on the basis that temporal or spatial locality of reference will lead to future memory accesses being satisfied from the cache [Smi82]. A common assumption in many multiprocessor algorithms is *sequential consistency*, which entails that all writes by one processor will be seen by all other processors in the same order as they were issued [Lam79]. A multiprocessor will have several caches, and coordinating processors will store overlapping regions of main memory in the course of reading from or writing to the same address. As a result, caches in a multiprocessor must employ additional *coherency* mechanisms to present a consistent view of memory to all processors. The simplest such mechanism involves broadcasting the addresses of write locations to all other caches, in order to invalidate any data that is cached at that location in the other caches, but this approach achieves poor performance for more than two processors [Smi82]. Tang refined this scheme by allowing caches to declare cache lines to be private to a single cache or shared between caches, which allows writes to private cache lines without a bus transaction [Tan76]. Various further refinements have been proposed and implemented; Archibald and Baer surveyed and

---

2However, in his 1977 survey [Ens77], Enslow does not discuss the Gamma 60, which was developed in the same period. Enslow’s focus on crossbar interconnects may suggest that he did not consider the Gamma 60 interconnect to be sufficiently parallel to be a “true” multiprocessor.
comparatively evaluated several different coherency schemes in a survey [AB86]. In general, however, these coherency schemes are *snoopy*: they rely on the ability of each cache to monitor a shared bus for invalidation messages.

As the number of processors grows, it becomes impractical for all processors to access memory through a single shared bus. As a result, multiprocessor architectures moved from symmetric to *non-uniform memory access* (NUMA). In a NUMA architecture, each processor—or group of processors—is directly connected to a portion of the total system memory with fast access, and indirectly connected to other processors’ memory with slower access [PH94, pp. 619–620]. The BBN Butterfly was an early NUMA system in which cache management and shared memory coordination had to be performed explicitly by the programmer [LSB88]. The Stanford DASH multiprocessor had a cache-coherent NUMA (ccNUMA) architecture, which used a hardware-managed *directory* to identify the processors on which a given memory location was cached [LLG+92]. Modern commodity multiprocessors use a similar design: the AMD Opteron “Magny-Cours” supports 12-cores on a single socket, with a hybrid directory protocol (known as HyperTransport-Assist) that provides cache coherency between sockets [CKD+10].

At the hardware level in a multiprocessor, cache coherency is provided by sending messages between processors, in order to present the illusion that the distributed memories form a single shared resource. However, neither cache coherency nor shared memory is strictly necessary for parallel operation: the underlying message passing hardware can be used directly for coordination. Recall that a MIMD machine with distributed memory is known as a *multicomputer*. The largest multicomputers have far more processors than the largest multiprocessors [TOP11], and hence have the potential to achieve greater parallelism.

The Inmos Transputer formed the basis of many early message-passing based multicomputers. A transputer was “a single VLSI chip integrating processing, memory and communication hardware” [Hey90]. Multiple transputers could be connected together to form a multicomputer, and each transputer was capable of executing multiple independent instruction streams using a hardware scheduler. Coordination between transputers was achieved by rendezvous: if process $P$ wanted to send a message to process $Q$, $P$ would issue a blocking output instruction on the link leading to $Q$. When $Q$ then issued a blocking input instruction on the link coming from $P$, the two processes would be synchronised, and the message could be copied from $P$ to $Q$ [MS88]. The transputer was designed to be installed in reconfigurable topologies; however, the relatively modest amounts of CPU power and memory on an individual transputer were soon exceeded by commodity sequential processors [IC99].

The idea of message-passing between modestly-powerful processors survives in the IBM Blue-Gene supercomputer [A+02]. BlueGene is a *massively-parallel processing* (MPP) multicomputer, in which up to $2^{20}$ cores (in the latest model, BlueGene/P [IBM08]) are connected using five distinct network topologies: a three-dimensional torus, a broadcast/aggregation tree and a barrier network are used to support common parallel algorithms, while two gigabit Ethernet networks provide control connections and access to shared storage [A+02]. The torus network
is exposed as a memory-mapped I/O device, which allows 128 bits to be read from or written to the network queues at once. The bulk of the network, including queueing, routing and flow control is implemented in hardware [BCC+03].

As the number of cores on an individual chip increases, cache coherency may become too expensive to maintain. As a result, Intel has developed an experimental 48-core processor, called the Single-chip Cloud Computer (SCC) [H+10]. The SCC contains 48 Intel P54C (second-generation Pentium®) cores, arranged in a six-by-four two-dimensional mesh of tiles, with two cores per tile. The P54C cores use 32-bit physical addresses, which are extended to 36-bit “system” addresses by a hardware lookup table, enabling a total system memory of 64 GB³. Unlike the 48-core AMD Opteron “Magny-Cours” described above, the SCC does not provide coherent access to shared memory, and therefore coordination must be performed in software. To enable this coordination, the SCC provides a hardware message passing buffer that enables low-latency communication between cores via an on-chip network.

2.1.3 Local-area networks

A computing cluster is a collection of computers connected by a local-area network for the purpose of executing computations in parallel. The distinction between multicomputers and computing clusters is not well defined; in this chapter, a computing cluster refers to processors connected by an asynchronous network, such as Ethernet. As a result, the message latency in a cluster is higher than in a multicomputer, and reliable message delivery cannot be assumed. Despite these drawbacks, computing clusters are attractive, because they can be built from cheaper commodity parts than those used in multicomputers.

A Beowulf cluster is the link between the multicomputers of the previous subsection and computing clusters. Beowulf clusters are built from commodity computing and networking hardware, and typically use open-source software (such as the Linux operating system and GNU C compiler), in order to achieve a low price-to-performance ratio [Ste02, pp. 2–3]. A Beowulf cluster can use user-space implementations of parallel runtimes, such as MPI [Mes94] or PVM [GBD+94] to provide a programming environment that is compatible with more-expensive multicomputers. However, while the programming model is the same as a multicomputer, the performance characteristics of a Beowulf cluster can be greatly different: programs that perform frequent barrier operations will tend to perform poorly when compared to running on a machine that provides a dedicated barrier network, such as BlueGene [A+02].

In 1995, Anderson et al. introduced the network of workstations (NOW) concept [ACP+95]. They observed the trend towards using commodity processors in large-scale multicomputers, and observed that using commodity workstations could solve other problems in parallel computing. In particular, most existing multicomputers are optimised for CPU- and RAM-intensive

³However, the 32-bit limitation of the P54C cores implies that an individual core may only address 4 GB of RAM at once.
workloads with disk I/O being limited to process loading and output collection [A⁺02, IBM08].
By contrast, in data-intensive workloads, the total data set is too large to store in the aggregate RAM, and so efficient disk I/O is necessary to access the data. The NOW vision includes a distributed version of a redundant array of inexpensive disks (RAID) [PGK88], whereby a distributed storage layer controls the disks in the cluster in order to achieve a higher aggregate throughput than a single disk (or a single RAID array). This approach led to a network of workstations becoming the world record holder for sorting a then-large amount of data (6 GB in under one minute) [ADADC⁺97].

Many commercial data centres have implemented a descendant of the Beowulf and NOW ideas. Google’s cluster architecture is designed to optimise the price-to-performance ratio, and uses thousands of rack-mounted x86-based servers with relatively inexpensive hardware. In particular, Google does not strive for high reliability using sophisticated hardware, and instead uses software-based replication to tolerate inevitable failures [BDH03]. Likewise, Microsoft uses commodity components in its search clusters, which are controlled by the Autopilot cluster management software. Autopilot uses replication and checksumming techniques to maintain fault tolerance in the face of unreliable hardware [Isa07]. The data-centre architectures are examples of what Stonebraker termed shared nothing, as “neither memory nor peripheral storage is shared among processors” [Sto86]. To coordinate between machines in a data centre, distributed coordination services are used. For example, Google uses the Chubby distributed lock service to provide mutual exclusion and a small amount of reliable distributed storage, which therefore also allows it act as a name server [Bur06]. The ZooKeeper coordination service, in use at Yahoo!, shares similar goals to Chubby, but can be used to implement a wider variety of primitives, such as group membership, rendezvous and barrier synchronisation [HKJR10]. These coordination services are intended for storing small data values, such as system configuration parameters, and do not scale to store terabytes of data. Therefore, in order to store large amounts of data, systems such as the Google File System (GFS) [GGL03], the Hadoop Distributed File System (HDFS) [SKRC10] and BigTable [CDG⁺06] have been developed. In each of these systems, there is a single master (or small number of masters) that stores metadata about the locations of data blocks, and multiple servers that store the data and serve it to clients. Although this leads to an architecture with a single shared component, the master only participates in metadata transactions. The data path is direct between the client and the servers, which allows this architecture to scale to a large number of machines.

Cloud computing is a recent development that has roots in the long-held ideal of utility computing. The idea of utility computing was introduced by John McCarthy in 1961, when he spoke of computing becoming a utility in the same way that telecommunications is a utility [GA99, p. 1]⁴. Cloud computing is a realisation of this idea, whereby corporations with an excess of data-centre capacity provide use of this capacity as a service to their customers [AFG⁺10].

Foster and Kesselman proposed a similar idea in the context of grid computing [FK98]. However, the realisation of grid computing has focused on widely-distributed systems [FKT01], and they are accordingly discussed in §2.1.4.
As a result, many services can be classed as “cloud computing”, including: collocated hosting (hardware as a service), virtual machine leasing (infrastructure as a service), higher-level programming APIs (platform as a service), and applications such as email or productivity applications (software as a service) [YBS08]. For the purposes of parallel computing, infrastructure as a service is the most common variant of cloud computing, because virtual machines give full control of the operating system to the untrusted customer and statistical multiplexing allows them to be provisioned on-demand for a short period of time [RPM+99]. As a result, cloud computing can support the same systems and programming models as a dedicated data centre, without the fixed costs of building such a data centre [AFG+10].

2.1.4 Wide-area networks

Traditionally, parallel computers were very expensive, and so many projects have investigated providing access to these resources across a wide-area network (WAN). The spirit of this movement is summed up in Wulf’s description of a “national collaboratory”, in which the advent of the Internet could allow scientific data, instruments and processing capabilities from different institutions to be shared nationally between those institutions [Wul93].

Smarr and Catlett suggested the idea of parallel computing across wide-area networks in their 1992 paper on metacomputing [SC92]. The idea of metacomputing was to connect together distributed computing resources in such a way that they could be programmed as a single metacomputer. In 1992, the state of the art in metacomputing was a local-area network connecting massively-parallel, vector processing and superscalar multicomputers and multiprocessors at the National Center for Supercomputing Applications (NCSA). However, Smarr and Catlett anticipated that WANs would soon become fast enough to support these applications in a widely-distributed setting.

A concrete implementation of metacomputing was developed in the context of grid computing. The aim of grid computing is to provide a platform for “coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations” [FKT01]. The main contribution of grid computing has been the Globus middleware and Open Grid Services Architecture (OGSA), which provide a standard interface for authentication, authorisation, resource allocation, job scheduling, file transfer and storage in a grid [FKNT03]. Wide-area coordination in a grid is achieved using calls to web services, and high-throughput data transfer uses technologies such as GridFTP [Hey03, p. 126]. It is interesting to note that grid computing and cloud computing are converging: in order to support untrusted applications, virtualisation-based approaches such as virtual clusters [FFK+06] and virtual workspaces [KFFZ05] have been integrated with grid middleware.

Peer-to-peer computing is an alternative approach to widely-distributed parallel computing, in which peers (independent computers) share resources to provide a distributed service [RD10]. The original peer-to-peer systems allowed volunteers to provide storage or computational re-
sources as part of a distributed system, for example for scientific computing or file sharing. These original systems used centralised coordination, whereby the peers would contact a central controller in order to obtain directions or query global system state. The advent of overlay networks enabled fully decentralized operation: in this model, the peers self-organise into a network by creating application-level links between one another, and coordinate by sending messages across the application-level links. One of the challenges is maintaining reliability in the face of churn—the constant arrival and departure of peers [SGG03]. For example, Chord is a distributed hash table (DHT) that provides key-value storage in a decentralised peer-to-peer network, and is robust to nodes joining and departing [SMLN+03]. In terms of applications, Huebsch et al. developed PIER, which is a peer-to-peer database [HHL+03] built on top of the Content-Addressable Network DHT [RFH+01]. Relatively little research has been carried out on general-purpose peer-to-peer computation, although the Triana middleware is able to coordinate parallel jobs using a peer-to-peer overlay [CGH+06].

2.2 Parallel programming models

Whereas the previous section showed the various mechanisms that can be used to coordinate parallel execution; this section surveys the programming models that can be built on top of these mechanisms. The first techniques that I will consider use shared memory for coordination, making them primarily suitable for use on multiprocessors only (§2.2.1). Explicit message passing is typically used on multicomputers, though it can also be used in any networked system (§2.2.2). Higher-level programming models can be built on top of either shared memory or explicit message passing, and I will discuss data-flow programming (§2.2.3) and declarative programming (§2.2.4).

2.2.1 Shared memory coordination

Shared memory parallelism is achieved by executing multiple threads within a single process. A thread is a “single sequential flow of control”, which can be implemented with an instruction pointer and a private execution stack [Bir89]. A process may contain multiple threads, which share the same address space. Parallelism is then achieved by assigning more than one thread from a multithreaded process to different processors in a multiprocessor.

Coordination in multithreaded programs is typically achieved by different threads reading from and writing to one or more shared memory locations; cache coherency (§2.1.2) is the underlying mechanism used to communicate values between different processors. However, since a processor can read from a memory region as another processor simultaneously writes to the same

---

5Note that a multithreaded process is not necessarily parallel. Multithreading can be used to overlap computation and I/O, as a multithreaded process can continue to make progress in one thread, while another thread is blocked waiting for I/O to complete.
region, care must be taken to ensure that the state of that memory region is consistent at the application level\(^6\). An intuitive correctness property is *linearizability*, meaning that a concurrent or parallel computation is semantically equivalent to a legal sequential execution [HW90]. A conservative way to ensure linearizability is to protect code containing accesses to shared regions with a *critical section* or *mutual exclusion lock* (mutex), which ensures that only one processor can execute that code at any one time. Dijkstra offered the first solution to the mutual exclusion problem based on two shared arrays [Dij65], though a modern implementation such as the Linux fast userspace mutex (futex) uses a combination of atomic instructions and system calls to avoid a polling loop in blocked threads [FRK02].

Brinch Hansen was one of the first to consider programming models for multithreaded programs, by extending the syntax of the sequential Pascal programming language [BH72]. To create parallel threads, the `cobegin...coend` block was used to denote a list of statements that should execute in parallel (the syntax was based on a similar construct that Dijkstra proposed for Algol 60 [Dij68]). Critical sections for a shared variable were denoted using the `region` statement, which would exclude simultaneous regions referring to the same variable. Coordination between threads was achieved with the `await` statement, which would block a thread (in a critical section) until a boolean expression became true; the expression would be re-evaluated every time a different thread exited a critical section on the same variable. Since this would lead to many re-evaluations, Hoare refined this model with the *condition variable*, which provides an explicit `signal` operation [Hoa74], and this feature is present in modern threading libraries [The08, PGB+05].

**POSIX threads** (Pthreads) is a low-level implementation of multithreading for POSIX-compliant operating systems (including Linux and BSD) [The08]. Pthreads is implemented as a library, which means it can be invoked from code written in any language; however, this means that Pthreads lacks some of the benefits of Brinch Hansen’s language-level approach. For example, mutual exclusion is implemented with the separate `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions, which makes it challenging to ensure at compile-time that calls to the lock and unlock functions are paired [ECH+01].

The Java programming language was designed to support multithreading [PGB+05]. As a result, it is able to support some of the structured multiprogramming concepts that Brinch Hansen proposed [BH72]. In Java, mutual exclusion is enforced on code within a `synchronized` block, and every object contains a lock and a condition variable. Java also includes a wide range of *concurrent collections*, including a thread-safe hashtable that supports multiple readers and concurrent writers, and a blocking queue that supports producer-consumer algorithms [PGB+05, §5.2].

It is also possible to write a multithreaded program with implicit thread creation and synchronisation. OpenMP is a shared-memory programming model that was originally designed to

---

\(^6\)Cache coherency does not imply application-level memory consistency, since application-level data structures may be larger than the unit of cache coherency (viz. a cache line).
parallelise the bodies of loops in C or Fortran programs. To parallelise a loop, it is annotated with an OpenMP compiler pragma [Ope08]. A suitable loop will have no loop-carried dependencies: for example, this loop can be parallelised trivially

```
for (i = 0; i < N; ++i) {
    a[i] = b[i] * c[i];
}
```

because the sets of memory locations read and written by each iteration are non-overlapping. On the other hand, this loop cannot be automatically parallelised without significant effort

```
for (i = 1; i < N; ++i) {
    a[i] = b[i] * a[i-1];
}
```

because each iteration depends on the result of the previous iteration—i.e. it contains a loop-carried dependency [CDK+01, §3.5]. OpenMP also supports parallel blocks [Ope08]—which are similar to `cobegin...coend` regions [BH72]—and has recently added support for parallel tasks [ACD+09], which I will discussed in more detail in Section 2.3.

### 2.2.2 Explicit message passing

Multicomputers and computing clusters do not have shared memory, so all of their parallel programming models are implemented on top of explicit message passing. This model has two fundamental operations:

- `send(dest, buffer, n)`
  - Sends $n$ bytes from the given buffer to the given destination processor, dest.

- `receive(src, buffer, n)`
  - Receives $n$ bytes from the given source processor, src, into the given buffer.

Several variants of these fundamental operations can be implemented, including asynchronous, non-blocking and buffered versions of send and receive; non-deterministic receive from any source; and optimised collective operations, including broadcast, reduce, scatter and gather. The Message Passing Interface (MPI) is a library that defines an interface for explicit message passing, including all of these operations [Mes94]. Implementations of MPI may optimise for the interconnect topology connecting the processors on which they are run, so operations such
CHAPTER 2. BACKGROUND AND RELATED WORK

(a) Computation

(b) Coordination

Figure 2.3: Conway’s Game of Life can be programmed using message passing. (a) In each round, the computation phase involves iterating over the local cells, to update their value. (b) After each round, each processor swaps halos (edge elements) with each of its eight neighbours.

as nearest-neighbour send/receive and global broadcast/reduce will be efficient on a machine such as the BlueGene, which provides torus and tree interconnects [A+02].

MPI is a versatile, low-level programming model for message-passing computation. The programming model typically assumes a single program, multiple data (SPMD) style of programming, in which all processors execute the same basic function, but use the process rank (ID) to perform different instructions. Figure 2.3 shows how message passing primitives can be used to compute multiple rounds of a one-dimensional version of Conway’s Game of Life [Gar70]. In this example, each processor acts on a contiguous partition of the problem space. After each round, each processor exchanges messages with its eight horizontal, vertical and diagonal neighbours, in order to perform a halo swap, which allows it to access the cells that adjoin its partition in the following round.

Many programming languages include first-class support for message passing. The Transputer was originally programmed in Occam, which provides language constructs for sending and receiving messages on a named channel [RH88]. Many concurrent languages are built on Hewitt et al.’s Actor model, in which all behaviour “can be defined in terms of one kind of behavior: sending messages to actors” [HBS73]. Erlang is designed for reliable execution with large

7In a shared memory implementation, these messages would not be necessary, because the processor could simply read from the array elements written by other processors. When a processor reads elements that have been written by other processors, this action will cause cache coherency messages to be exchanged between the processors.
numbers of processes, and its concurrency model is based completely on copying messages between processes, as there is no shared state [Arm07].

Attempts have been made to reconcile message passing with conventional single-threaded control flow. Birrell and Nelson described the implementation of a remote procedure call (RPC) facility, which presents to the caller the appearance of synchronously calling a function, while transparently marshalling the arguments and return value between the client and server [BN84]. One limitation of RPC is that the server is statically bound to a single network location. The Emerald programming language incorporated the ideas from RPC in its run-time system, which provided a distributed and mobile object model [BHJL07]. As a result, Emerald provided mechanisms for creating objects, moving objects between network locations, and transparently invoking methods on those objects. This mechanism was the basis for modern distributed component systems, such as Java Remote Method Invocation (RMI) [Wal98] and CORBA [Obj08].

Given the message passing operations, one might ask whether it is possible to implement shared memory algorithms in terms of messages (or vice versa). Lauer and Needham identified a duality between message-passing and “procedure-oriented” (shared-memory) operating systems [LN79]. In their model, a message-passing system that provides the send, receive-from-any and receive-from-one operations is the dual of a shared-memory system that provides monitors and condition variables.

Valiant introduced bulk synchronous parallel (BSP) as a “bridging model” for general-purpose parallel computation [Val90]. A BSP program is divided into supersteps, during which all processors execute independently in parallel, and after which there is all-to-all communication between the processors. The programming model is not coupled to the underlying topology: it can be implemented on a message-passing multicomputer, but it can also efficiently simulate a concurrent-read concurrent-write (CRCW) parallel random access memory (PRAM), which models a shared memory [Val88]. Hence, at least in theory, any shared memory program can be rewritten as a BSP program, which can run on a message-passing machine. However, as Valiant observes, the performance will be determined by several factors, including the bandwidth and latency of the interconnect.

There exist practical approaches that attempt to reconcile shared-memory programming models with a message-passing system. Distributed shared virtual memory (DSVM) is a systems-level approach to simulating a single shared memory. Recall from §2.1.2 that cache coherency protocols are implemented using hardware messages between caches in a single multiprocessor. The same principle can be applied to a distributed system, although to make it practical, the unit of data transfer must be larger. Li and Hudak’s IVY system creates a shared memory across a network of workstations by modifying the implementation of virtual memory to fetch pages across a network [LH89]. A processor faults on the first instruction that attempts to write to a page, which invokes the DSVM handler. The faulting processor obtains the page by sending a message to the “manager” processor for that page. The manager maintains a directory of the processors that hold a copy of the page, and sends a message to each processor to invalidate the
copy. The manager finally sends to a message to the “owner” of the page (i.e. the last processor that wrote to the page), instructing it to send the contents of the page to the faulting processor. Li and Hudak demonstrated several variants of this algorithm, which principally differ in how the manager of a page is assigned. Although a systems-level approach theoretically allows unmodified shared-memory programs to run in a distributed system, the performance of an equivalent message-passing implementation is generally much faster [LDCZ95].

The translation from memory accesses to message passing can also be achieved at the language level. For example, the High Performance Fortran (HPF) programming language supports “directives” on array declarations that enable an array to be partitioned between processors along each of its dimensions [Hig93]. Once an array has been partitioned, collective operations (such as the Fortran array operators, and FORALL loops through all of the elements of an array) are compiled into the appropriate message exchanges [BZ99]. Unfortunately, the adoption of HPF was hampered by poor compilers, which were not able to achieve good performance across a range of different systems [KKZ07].

The idea of partitioning data structures across multiple distributed processors is carried on in the partitioned global address space (PGAS) languages. PGAS languages include Unified Parallel C [CDC+99], X10 [CGS+05], Fortress [ACH+07] and Chapel [CCZ07]; and each provides a global address space that can be implemented on top of message passing. Unified Parallel C is an extended version of C that provides the facility to annotate variable declarations with a shared modifier, a upc_forall loop that resembles HPF’s FORALL, and functions for performing synchronisation based on global single- and split-phase barriers. Fortress is a new language based on Fortran, which supports HPF-style distributed arrays and the ability to dynamically spawn threads in a “region”, which may correspond to a processor, a group of processors or some other domain of locality. Chapel deliberately eschews the “fragmented” SPMD programming model, in order to support nested parallelism using a cobegin...coend syntax that resembles Brinch Hansen’s extensions to Pascal [BH72]. Finally, X10 extends the PGAS concept to the Java programming language, with a focus on “non-uniform cluster computing”, which is defined as hierarchical parallelism in clusters of multiprocessor machines.

The principal drawback of message passing is that the sender of a message must know the identity of the message recipient. This is not a problem for many supercomputing algorithms, since the topology is static and recipient addressing is primarily relative (i.e. to the nearest neighbours in a torus) or collective. However, in a less-coupled system, such as a data centre, it may not be feasible to maintain accurate details of cluster membership at every node, due to nodes failing and coming online. Another drawback is that the programming models described in this subsection assume that the data is stored in memory, which does not scale to handle large data sets that are primarily stored on disk in a distributed file system [GGL03, SKRC10]. As a result, there has been substantial interest in programming models that decouple program

---

8For example, a dimension may be declared with the following distributions: BLOCK (equal-sized partitions of contiguous elements on the same processor), CYCLIC (K) (K contiguous elements on the same processor, then cycling through the processors) or * (all elements on the same processor).
CHAPTER 2. BACKGROUND AND RELATED WORK

Figure 2.4: A simple data-flow graph for computing the expression $x^2 + yz$. Each circle is a node, and the edges indicate the order of evaluation. Here, the two multiplications can execute in parallel, followed by the addition.

execution from the details of where the programs run and how the data are transferred. The following subsections describe two such programming models.

2.2.3 Data-flow programming

Data-flow programming uses a graph-based programming model to define the operations that are applied to data as it flows from input to output. Davis and Keller define data-flow languages to be *applicative*, which means that they are defined in terms of pure function application; therefore, execution has no side-effects and all coordination is explicit in the data-flow graph [DK82].

A data-flow graph is represented as a directed graph of *nodes*, which correspond to executable operations, and *edges*, which correspond to data dependencies between the operations. Figure 2.4 shows a simple example of a data-flow graph. The nodes are pure functions, which take the input data on incoming edges, perform some transformation, and produce output data on the outgoing edges. Data-flow graphs are well-suited to expressing parallel programs, because—unlike the programs written in the imperative models discussed above—there is no implicit ordering between sequences of operations: an operation can run as soon as all of its dependencies are fulfilled.

In a data-flow graph, data may be modelled using the *token model* or the *structure model* [DK82]. In the token model, data flow along the edges of the graph as a stream of discrete tokens; each node consumes one token on each of its incoming edges, and produces at least one token on at least one of its outputs. Nodes may be conditional: for example, a *selector* node forwards one of its (e.g. two) data inputs to its single output based on the value at another (e.g. boolean) input; a *distributor* is the equivalent single-input multiple-output node. These constructs may be combined to perform unbounded iteration, by introducing a cycle in the graph. Alternatively, a node may recursively replace itself with a subgraph, as long as the recursive expansion is lazily
evaluated when a token arrives at the recursive node\(^9\).

In the structure model, the stream of tokens on each arc is replaced by a single data structure (which may however be interpreted as a stream of tokens). The principal difference between the token model and the structure model is that, in the structure model, the behaviour of the data-flow graph may be expressed as a (recursive) function of the input structures. In the token model, the behaviour is a function on the histories of the token streams. For example, consider a token on an edge that forms a cycle: it is not obvious (from the token itself) which input tokens resulted in its production, and so reconstructing the token would require additional history information to be retained. By contrast, in the structure model, a node output can be reconstructed from the unique inputs to that node. According to Johnston \textit{et al.}, the token model became predominant in data-flow programming [JHM04]; however, the difficulty of reconstructing a history has implications for fault tolerance, and motivates the use of the structure model for the system described in later chapters of this dissertation.

Although data-flow programs can be represented by graphs such as those in Figure 2.4, a data-flow \textit{language} need not be graphical\(^10\). Ackerman outlined the properties of a data-flow language [Ack82]:

1. Operations must not cause side effects, i.e. they must be pure functions.
2. All instruction scheduling constraints must be captured in the data dependencies.
3. Variables may only be assigned once (the \textit{single assignment} property).

The second and third properties are corollaries of the first. However, the single assignment property has consequences for the syntax that may be used in iteration. For example, the following C fragment would not be a valid data-flow program:

```
1  y = ...;
2  do  {
3      x = y;
4      y = f(x);
5  } while(x != y);
```

This program does not obey the single assignment property, because \(x\) and \(y\) are assigned multiple times. If the same storage location were used for those variables, data-flow parallelism

\(^9\)The execution model described in Chapter 3 uses a similar technique based on rewriting data-flow graphs to achieve computational universality. However, the expansion is constructed by evaluating the result of executing a node, rather than interrogating its definition.

\(^{10}\)Of course, this does not preclude the existence of \textit{data-flow visual programming languages} (DVPLs), which use a graphical user interface to manipulate programs as graphs [JHM04].
would lead to potential hazards in accessing those variables. Many data-flow languages allow
names to be reused while respecting the single assignment property, by prefixing an annotation
to the offending assignments:

```c
!y = ...;
!do {
    new x = y;
    new y = f(x);
} while(x != y);
```

This second program is a valid data-flow program, if we assume that the `new` modifier implicitly
creates a new storage location for the assignment, and subsequent uses of the variable name refer
to the most recently-constructed instance of the variable.

In the original implementation of data-flow languages, each node in the data-flow graph corre-
sponded to a hardware-level instruction, such as an addition or a multiplication. The data-flow
community believed that the von Neumann model was ill-suited to parallel execution, due to
the overheads of synchronisation; to quote Agerwala and Arvind, writing in 1982, “Very few
machines based on this [parallel von Neumann] approach have been designed so far; those
that have been built have not had significant success in exploiting parallelism” [AA82]. As a
result, there was significant interest in the application of data-flow techniques to computer ar-
chitecture, for example in the Manchester Prototype Dataflow Computer, described by Gurd et
al. [GKW85]. However, the von Neumann architecture has been more successful, for many of
the reasons advanced by Gajski et al., also in 1982 [GPKK82]: in particular, the serial perfor-
mance of a data-flow computer is poorer than a von Neumann machine, due to the logic that is
necessary to identify whether a data-flow node is runnable.

Bic proposed a compromise between the data-flow and von Neumann approaches [Bic90]: in-
stead of treating each instruction as a data-flow node, let the nodes represent simple processes or
tasks, which maintain the same side-effect-free property as instructions in a data-flow language.
A high-level analogue of a data-flow machine could then schedule the tasks across multiple
processors, based on the state of the data-flow graph. Many systems have adopted this model,
including the system described in this dissertation. More details of coordination schemes for
parallel tasks are discussed in Section 2.3.

### 2.2.4 Declarative programming

In the foregoing programming models, the programmer must describe exactly how a result
is computed. **Declarative** parallel programming models are based on the intuition that, if a
programmer simply specifies what result is desired at a high level, the system can devise a
parallel method to produce that result [Llo94].
Declarative languages have enjoyed great popularity in the database field. The Structured Query Language (SQL) is a declarative language that can be used to specify queries over data stored as sets of relations [EM99]; every major relational database management system (RDBMS) supports a dialect of SQL. SQL is based on Codd’s relational algebra, which was intended to hide details of “how the data is organized in the machine (the internal representation)” [Cod70]. The relational algebra (and hence SQL) is based on first-order logic, and includes expressions that filter relations based on a predicate, project attributes from a relation and denote the Cartesian product of two relations (so that they may be joined). The relational algebra is not universal, because it cannot represent a query that computes the transitive closure of a relation; Aho and Ullman observed that it can be made more expressive by adding a least fixed point operator, which enables recursive computation [AU79], and an equivalent syntax for recursive queries was eventually been added to SQL [EM99].

SQL is amenable to parallelism, because its small number of operations have well-understood parallel implementations [DG92]. For example, filtering and projection can be applied independently to each relation in parallel, which allows the data to be partitioned between many processors; parallel (equi-)joins of two data sets can be achieved by repartitioning both data sets based on the key attribute that is used in the join condition, and performing local joins between the two subsets of the data in each partition of the key space. SQL is also well suited to query planning, which converts a SQL query to a tree or DAG of relational algebra operators, and reorders the operators (while preserving query semantics) to reduce the amount of computation and data motion: for example, filters are moved as early as possible, in order to reduce the amount of data processed by subsequent operators.

Declarative queries may also be supported within imperative programming languages. The Language INtegrated Query (LINQ) framework supports SQL-style query comprehensions as a syntactic extension to the C♯ and Visual Basic .NET programming languages [BMT07]. A LINQ query comprehension may be computed over any enumerable data structure, which achieves Codd’s goal of abstracting the precise data representation. The principal difference from SQL is that a LINQ query comprehension is statically typed, which simplifies the handling of structured data types and polymorphic objects, and allows the comprehension to invoke methods on those objects in a type-safe manner. LINQ supports a variety of “query providers”, which implement query execution. The simplest query provider is LINQ-to-Objects, which translates query comprehensions into calls to generic iterator functions that perform operations such filtering, transformation and grouping. Parallel LINQ (PLINQ) is a query provider that partitions the data between threads in the same process and pipelines intermediate elements between operators, to achieve parallelism on a multiprocessor [LSB09]. DryadLINQ is a query provider that executes queries in parallel on data that is partitioned across a computing cluster [YIF+08]; it is based on the Dryad execution engine [IBY+07], which I will discuss further in the following section.

11SQL also specifies statements that modify and define relations; however, this discussion focuses on queries because these typically require more computation, and are more amenable to parallelisation.
Datalog is a declarative logic language: a Datalog program is a set of “facts” (analogous to relations in a database) and “rules” (analogous to computed views derived from the facts) [CGT89]. Datalog arose as an alternative to SQL (prior to the addition of recursive queries) for data retrieval, because it is more expressive, yet all pure Datalog programs are guaranteed to terminate. For example, the transitive closure of a rule $r$ can be straightforwardly expressed with two Datalog rules:

1. $rPlus(x, y) :- r(x, y)$
2. $rPlus(x, y) :- r(x, z), r(z, y)$

However, pure Datalog is not universal, because it has no negation operator and therefore it can only compute queries that are strictly monotone [KV90]. Various extensions to Datalog and its operational semantics have been proposed, which range from limited support for negation (using stratification, which prohibits cyclic rules where the cycle contains negation) to full Turing-completeness [AV91].

Recently, there has been renewed interest in Datalog in the context of distributed systems. Loo et al. developed the P2 system, which uses their OverLog variant of Datalog to implement peer-to-peer overlays in a declarative manner [LCH+05]. OverLog extends Datalog with negation, “streams” (virtual rules that do not correspond to stored tables) and “location specifiers” (annotations on rules that specify at which overlay node a tuple should exist). Loo et al. used the initial version of OverLog as the programming language for the P2 declarative overlay system [LCH+05]. Alvaro et al. have used OverLog to build several distributed systems, including a declarative implementation of distributed consensus [ACC+10b] based on Paxos [Lam98], and a distributed software stack for data analysis [ACC+10a] based on Hadoop [Had]. To date, most use of OverLog has been in implementing back-end coordination systems, such as peer-to-peer overlays and distributed execution engines, and it has not found great adoption in application-level programming [ACC+10a]; consequently, Alvaro et al. implemented their follow-up language, Bloom, which is inspired OverLog, as a domain-specific language that is embedded in the Ruby scripting language [ACHM11].

The common theme of the previous two sections has been that the “pure” programming models of the data-flow and logic programming languages become more useful when they are embedded in a “conventional” imperative programming (cf. Bic’s process-level data-flow [Bic90], DryadLINQ [YIF+08] and Bloom [ACHM11]). This thesis takes the same approach: the dynamic task graph execution model (Chapter 3) supports various different programming models (Chapter 5), including embeddings in existing functional and imperative languages.
CHAPTER 2. BACKGROUND AND RELATED WORK

2.3 Coordinating distributed tasks

Bic’s proposal for process-level data-flow parallelism [Bic90] can be seen as foreshadowing the advent of task-parallel distributed execution engines. In Chapter 1, I introduced the concept of a distributed execution engine as a system “that exposes high-level operations for performing parallel computations”. The predominant model of parallelism in a distributed execution engine is task parallelism, whereby processors work on independent tasks, which correspond to Bic’s notion of high-level processes that have no side-effects other than producing expected outputs. Although one could conceive of different execution models for a distributed execution engine (for example, based on explicit message passing between processors), this section and the remainder of this thesis focuses on task parallelism because it has been shown to scale to very large commodity clusters [DG04, IBY+07].

The basic system architecture for a distributed task-parallel system (and hence a distributed execution engine) is the task farm (Figure 2.5). In a task farm, the master maintains a queue of runnable tasks. A task farm contains one or more workers: when a worker becomes idle, it requests a task from the master, executes the task, sends the task result to the master, and then becomes idle once more\textsuperscript{12}. This architecture has several advantages. All coordination is between a worker and its master, which is simple to implement in either a shared-memory or a message-passing system. In particular, it naturally supports non-uniform task sizes or processor speeds, since each processor consumes tasks from the queue independently of all others\textsuperscript{13}. The throughput of a task farm can be improved by adding more workers. Since tasks are independent

\textsuperscript{12}The origins of the term “task farm” are somewhat obscure. The earliest explanation appears to be due to Bowler et al. [BBK+87], which describes the coordination between a master and a “slave” (worker) process.

\textsuperscript{13}Note that this only becomes true as the number of tasks becomes large. In smaller jobs, a slow-running processor or long-running task—known as a straggler—may have a large impact on the overall job completion time [DG04], and several strategies have been suggested to ameliorate this problem [ZKJ+08, AKG+10].

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{task_farm_diagram.png}
\caption{Basic task farm architecture. The master maintains a queue of tasks. The workers request tasks from the queue, process the tasks, and send results back to the master, which aggregates the results in an accumulator.}
\end{figure}
and have no side-effects, they can safely be re-executed in the event that a worker fails, which reduces the problem of (worker) fault tolerance to failure detection.

In a task farm, the master contains a *dependency resolver*: an abstract component that controls admission to the task queue, based on the initial job and any task results that have been received. The implementation of the dependency resolver determines the class of data-flow graphs (representing computations) that a distributed execution engine can support. This section surveys systems that employ increasingly-sophisticated dependency resolvers, in terms of the computations that they can support. Subsection 2.3.1 considers the simplest model, in which the dependency resolver is trivial, because all tasks are independent. Subsection 2.3.2 extends this to systems that allow a fixed dependency structure between tasks. Fixed dependencies can be generalised to arbitrary directed acyclic graphs (DAGs) of tasks, as discussed in Subsection 2.3.3. The dependency resolver may be an active component, which allows tasks to be added during a job: Subsection 2.3.4 considers how this may be used to support iteration, and Subsection 2.3.5 discusses a model in which tasks may dynamically fork new parallel tasks.

Figure 2.6 illustrates the data-flow graphs that each system supports.
2.3.1 Independent tasks

In the simplest task-parallel model, all tasks are independent, which means that the dependency resolver simply admits all tasks in the job to the queue. There is no data flow between tasks, so Figure 2.6(a) shows a data-flow graph with no edges.

Condor is a “cycle-stealing” task farm, which aims to harness idle processor cycles from workstations on a local-area network (with extensions to wide-area networks) [TTL05]. On a practical level, Condor is designed to run unmodified (or minimally-modified) applications using distributed resources, and it provides mechanisms that virtualise file system and console access so that a process running on a remote computer executes in a similar environment to the local computer. The main research contribution of Condor is flexible support for “matchmaking” between heterogeneous resources and heterogeneous tasks [RLS98]. The Condor matchmaking language allows resources and tasks to be specified in a structured yet flexible syntax, which allows the specification of hard and soft constraints; and the “matchmaker” attempts to schedule tasks on the machines that best meet the tasks’ requirements. Condor has been extended with a “flocking” mechanism that allows tasks to be scheduled in different LANs, which allows the construction of a wide-area cluster [ELvD+96].

The SETI@home project is an example of a wide-area task farm that uses independent task parallelism [ACK+02]. The distributed computing platform used by SETI@home was the basis for a generic middleware supporting independent task parallelism, the Berkeley Open Infrastructure for Networked Computing (BOINC) [And04], which underpins a wide range of volunteer computing projects. In SETI@home, volunteers around the world can donate the resources of their home computers to a distributed computation that is seeking evidence of extra-terrestrial intelligence in recordings taken from radio-telescopes. Each task corresponds to a time- and frequency-divided fragment of radio-telescope data, and the worker performs various digital signal processing operations over the data to identify improbable patterns in the data. As a widely-distributed system with untrusted participants, SETI@home must make several tradeoffs: for example, the size of a work unit must be large enough that it keeps the communication-to-computation ratio small (since WAN links have high latency and low bandwidth), but small enough to maintain a high probability that the task will complete before the worker leaves the system (since the workers are pre-empted when the local user resumes control of the volunteered computer). Furthermore, to counter the risk that a volunteer will modify the worker to report successful completion (and hence, accrue points for that volunteer on a public leaderboard) without actually processing the data, the system must distribute multiple copies of each task and cross-validate the results before considering the task to be completed.

The C♯ and Java programming languages contain support for running independent tasks on a multiprocessor. In C♯, the Task Parallel Library (TPL) allows developers to invoke tasks—specified as delegate functions or lambda expressions—in parallel using a thread pool [LSB09]. The concurrent extensions to Java version 5.0 provide the Executor interface and framework, which provide a similar service for objects that implement the Runnable interface [PGB+05,
§6.2]. Note that, since these facilities are implemented in a shared memory system, it behooves the programmer to ensure that the tasks do not have side-effects that could lead to race conditions between tasks, and hence possibly-incorrect results.

The computations that an independent-task system can perform are known as embarrassingly parallel. Most embarrassingly parallel algorithms perform a sweep through a large parameter space, such as the time and frequency domains in SETI@home, the initial conditions of a protein in Folding@home [SNPG01], or the random seed in a Monte Carlo simulation [MU49]. However, many parallel algorithms do not fit this model. For example, the distribution sort algorithms for sorting large amounts of data are not embarrassingly parallel, because they require two stages: a partitioning stage that groups records in the same key range on the same machine, and a local sort stage that sorts within each key range [Knu73, pp. 347–350]. The tasks in the second stage cannot run until the first stage has completed, which introduces a dependency between tasks. The following subsection introduces systems that can support a limited class of dependencies between tasks, which enables algorithms such as distribution sort to be computed in those systems.

2.3.2 Fixed data flow

In a fixed data-flow system, edges may be added to the data-flow graph (Figures 2.6(b) and (c)), but only in a constrained manner. In addition, there may be constraints on the task behaviour, such as the number of inputs or outputs, or the form of processing that the task performs.

Google’s MapReduce programming model and distributed execution engine has been hugely influential since its publication in 2004 [DG04]. The MapReduce programming model is based on the map() and reduce() (fold()) higher-order functions that are commonly used in functional programming languages\textsuperscript{14}. The input to a MapReduce program is a large file, containing a sequence of records. First, the map() function is applied to all of the records independently, in parallel tasks\textsuperscript{15}. Each invocation of the map() function generates zero or more key-value pairs as intermediate data. The MapReduce runtime then groups the intermediate data by key (e.g. using a range- or hash-based partitioning scheme [DG92]), and applies the reduce() function independently on each unique key and the bag of associated values. The data-flow graph (Figure 2.6(b)) is therefore a complete bipartite graph, comprising $m$ map tasks and $r$ reduce tasks. $m$ and $r$ are the only parameters under user control: $m$ corresponds to the number of partitions in the input data set, and $r$ must be selected by the user, based on the expected output size. A trivial invocation of MapReduce can therefore perform a distributed sort, as posited in the previous section: the map() function simply emits its input record unmodified, the runtime sorts by the key, and the reduce() function sorts within the group of values, if necessary.

\textsuperscript{14} Various forms of the map() function were available as standard in version 1.0 of LISP [MBE+60, p. 122]; the reduce() function is based on the higher-order reduction operator of APL [Ive62].

\textsuperscript{15} A typical MapReduce implementation will batch records into “chunks”, and process records sequentially within chunks, in order to mitigate task creation overheads [DG04, Had, RRP+07].
One limitation of the MapReduce model is that it reads a single input and writes a single output, which makes it challenging to implement joins of multiple data sets. To address this limitation, Yang et al. developed Map-Reduce-Merge, which adds a third round of “merger” tasks that runs after the reduce tasks [YDHP07]. A Map-Reduce-Merge job first performs two separate MapReduce jobs on two input files. The merger tasks read subsets of the reduce outputs from each job as necessary, in order to compute the required merge. In the simple case of an equi-join, assuming that the two MapReduce jobs partitioned the reduce key space identically, each merger task would retrieve one output partition from each job, then perform a local hash-join between the partitions. Map-Reduce-Merge also introduced the concept of building workflows of MapReduce jobs, which enables the construction of arbitrary acyclic data-flows, as discussed in the following subsection.

Other fixed data-flow systems have been proposed. For example, Yu et al. developed distributed execution engines for wavefront parallelism [YME+09], which can be implemented with a fixed data-flow graph. In wavefront parallelism, tasks are arranged in a regular $k$-dimensional mesh, in which each internal task depends on its predecessor along each dimension (Figure 2.6(c)). Data-flow starts at the origin task, which is located in one corner of the graph and has no dependencies, and proceeds across the mesh in a diagonal wavefront that grows as more tasks become runnable, then shrinks as fewer tasks remain in the opposite corner. The wavefront abstraction is well-suited to dynamic programming algorithms, which involve computing all of the elements in a $k$-dimensional array, where each element depends on its predecessor in each dimension [Bel54]. A parallel dynamic programming algorithm maps each task to a contiguous block of the array, and the data-flow involves passing the values on the leading edge of each block to the successor task. Moretti et al. implemented another fixed data-flow system for computing all-pairs jobs, which compute the Cartesian product of two data sets [MBH+10], and may be used to compute a general join over the data sets [Cod70]. In Moretti et al.’s implementation, the all-pairs abstraction was used to compute pair-wise similarity between faces in a large data set, for a biometric application.

Fixed data-flow systems can obtain large degrees of parallelism on the problems for which they were designed. However, they tend to be application specific, and they do not provide the ability to compose parallel jobs into a larger workflow. The next category of systems supports this model by representing jobs with a more general structure.

### 2.3.3 Acyclic data flow

In an acyclic data-flow model, a job is specified as an arbitrary DAG of tasks. This generalises the fixed data-flow model, because it allows each task to perform any computation internally,
with any finite number of inputs or outputs. Therefore, an acyclic data-flow graph can represent any fixed data-flow, and any finite composition of fixed data-flows. To support acyclic data-flow, the dependency resolver must be able to compute a topological order of the tasks, which can be computed for any DAG [Las61]. A DAG may admit many different topological orders, because some tasks are independent, and this leads to the opportunity for parallel execution.

The *make* build-management tool is a prototypical acyclic data-flow system, based on topological ordering [Fel79]. A Makefile allows developers to provide a declarative specification of how a project is built, by specifying production rules that indicate how a file can be created in terms of a list of dependencies. For example, consider how to build a C project that contains three C source files (*foo.c*, *bar.c* and *baz.c*), as depicted in Figure 2.7. Each source file can be compiled to an object file, depending only on the source file and a header file (*inc.h*). Finally, the object files are linked together to form an executable (*a.out*). Note that each compilation task is independent, and may execute in parallel: this observation has led to several concurrent and parallel versions of *make*, which can achieve speedup on a uniprocessor (by overlapping computation and I/O), multiprocessor or computing cluster [Baa88].

In 2006, Isard *et al.* published details of the Dryad distributed execution engine, which is used to power Microsoft’s Live Search engine\(^\text{17}\). Dryad is a successor to MapReduce, which is also designed for processing large files that are distributed across a commodity cluster. However, Dryad provides a more-expressive execution model than MapReduce, because it supports acyclic data-flow. This enables a single Dryad job to contain a composition of several MapReduce-style computations, which collectively execute in a fault-tolerant manner. A Dryad computation is built from a DAG, in which the vertices correspond to tasks, and the edges are “channels” be-

---

\(^\text{17}\) Live Search is presently known as “Bing”.

---
tween those tasks. In the original presentation of Dryad, programs were written using a domain-specific language for building DAGs that was embedded in C++ [IBY’07]. However, several higher-level programming models have been built on top of Dryad, including SCOPE [CJL’08] and DryadLINQ [YIF’08]. SCOPE is a variant of SQL, with additional support for inline C♯ user-defined functions, which can be transformed into a Dryad job [CJL’08]. DryadLINQ uses the query provider facility of LINQ (introduced in §2.2.4) to build Dryad jobs directly from a query comprehension embedded in a C♯ program [YIF’08]. Both languages provide a declarative programming model for distributed queries, though DryadLINQ goes further in terms of integrating the queries with an existing .NET application, because it can transparently invoke and be invoked from existing C♯ code.

Several projects have retrofitted support for acyclic data-flow onto the MapReduce execution engine. Pig Latin is a high-level scripting language for creating workflows of Hadoop MapReduce jobs [ORS’08], while Hive and its HiveQL query language have been designed to provide a relational-style database layer on top of Hadoop [TSJ’10]. Google’s FlumeJava is a high-level Java library that enables the specification of acyclic data-flows, which are then executed using Google’s implementation of MapReduce [CRP’10]. Each of these systems supports a more-expressive programming model (acyclic data-flow) using a less-powerful execution engine (fixed data-flow). In order to achieve this, some code must run outside the cluster to perform the necessary dependency resolution. This is less desirable than a system like Dryad, because Dryad has full knowledge of the job dependencies, which enables it to provide fault tolerance for the whole job by selectively re-executing tasks to reproduce missing values. By contrast, these external dependency resolvers must re-implement a second level of fault tolerance outside the cluster, and, since their only interface to the cluster is job submission, they cannot perform such fine-grained re-execution whenever necessary.

The acyclic data-flow systems are more powerful than their predecessors in this section, but there are many algorithms that they cannot natively support. In particular, because the data-flow is acyclic and static, they cannot express algorithms that contain unbounded iteration or recursion, which is the motivation of this thesis. Several systems have been proposed to address this limitation, as the following subsection discusses.

### 2.3.4 Iterative data-flow

Many potentially data-parallel jobs involve fixpoint iteration, which requires a more expressive execution model than those discussed in the previous subsections: in particular, supporting fixpoint iteration requires data-dependent control flow. In a von Neumann architecture, iteration is achieved with a conditional branch instruction; a data-flow architecture uses cyclic depen-

---

Note that, in Dryad, the job DAG may be rewritten to make performance optimisations—such as the speculative re-execution of straggler tasks or the construction of an aggregation tree for commutative and associative operators—but there is no interface for application code to modify the graph at run-time [IBY’07].
dependencies or recursive expansion (§2.2.3). This subsection surveys the approaches that have been taken along with their limitations.

Just as acyclic data-flow can be simulated in a fixed data-flow system, iterative data-flow can be simulated by adding a driver program that runs outside the cluster and performs the necessary data-dependent control flow. A driver program for iterative data-flow has the following structure:

```
input = ...;
x = ...;

do {
    submitJobToCluster(input, x);
    waitForJobCompletion();
    updatedResult = fetchResultFromCluster();
    converged = convergenceTest(x, updatedResult);
    x = updatedResult;
} while (!converged);
```

The driver program submits a job to the cluster, waits for and fetches the result, and evaluates a convergence criterion on it. All data-dependent control flow is performed in the driver program: based on the result of the convergence test, it might then submit another job. A driver program can be written against an existing non-iterative framework: for example, the Apache Mahout scalable machine learning framework uses driver programs to perform iterative computations using Hadoop MapReduce [Mah]. Similarly, a DryadLINQ query may be wrapped in a while loop to perform iterative computation on Dryad [YIF+08, IBY+07]. The main limitation of the driver program approach is that it does not support fault tolerance between jobs. The execution stack of the driver program contains critical state about the present iteration; therefore, if the driver program fails (for example, if it cannot access the cluster due to a network partition), the entire computation is lost unless custom fault tolerance routines are added to the driver program. Therefore, driver programs do not enjoy the transparent fault tolerance that motivates the use of task parallelism. Furthermore, even if a failure is successfully detected and handled, the driver program can only perform recovery by resubmitting entire jobs, which potentially wastes resources by re-executing successful computation.

The Spark cluster computing framework uses the driver program approach to perform efficient iterative computation [ZCF+10]. A Spark computation is written in the Scala programming language, using primitives that resemble DryadLINQ. The framework converts the computation into stages of independent tasks, which it schedules using task farming. Spark achieves greater efficiency than many frameworks for iterative computation, because it can cache intermediate results in memory, which improves the efficiency of repeatedly accessing large preprocessed data sets. Spark also incorporates the concept of a “resilient distributed dataset”, which enables
derived results to be recomputed between jobs in the event of failure. However, since it uses a
driver program to perform iteration, Spark is vulnerable to client failure.

Power and Li developed Piccolo [PL10], which is a programming model for data-parallel pro-
gramming that uses a partitioned in-memory key-value table to replace the reduce phase of
MapReduce. The in-memory tables provide a similar benefit to Spark’s intermediate result
cache. A Piccolo program is divided into “kernel functions”, which are applied to table par-
titions in parallel, and typically write key-value pairs into one or more other tables. Data ag-
gregation is performed by user-defined aggregation functions, which may be attached to the
key-value table, and are applied to new values as they are written into the table. A Piccolo
“control function”—i.e. a driver program—coordinates the kernel functions, and it may perform
arbitrary data-dependent control flow. Piccolo is implemented using MPI, and supports fault tol-
erance via user-assisted checkpointing (based on the Chandy-Lamport algorithm [CL85]). The
checkpointing support makes it easier to persist the state of the driver program in the event of
failure, although the user must still add logic to restart a computation from the checkpoint. One
limitation of Piccolo’s fault tolerance model is that the cluster size is fixed, which means that
recovering from a single machine failure would require obtaining a replacement machine to
restart from a checkpoint.

Valiant’s BSP is a fully-distributed execution model that supports iterative computation on top of
message passing [Val90], and it therefore represents a possible basis for an iterative execution
engine. Google’s Pregel [MAB+10] is a system based on BSP for executing iterative graph
algorithms (such as PageRank [PBMW99]) on very large data sets. The input to a Pregel job is
a large set of graph vertices that is partitioned between worker nodes, and stored in memory on
those nodes. Computation proceeds in supersteps, which involve mapping a “vertex method”
across the whole data set. A vertex method may send messages to neighbours in the graph,
modify the graph topology, and—crucially for iterative computation—vote to terminate the
computation. A Pregel computation terminates when all vertices have voted to terminate. Pregel
employs some concepts from MapReduce to improve scalability: for example, messages sent to
a vertex may be combined (reduced) by an aggregation function in order to decrease the amount
of data that is sent across the network. The main limitation of the Pregel programming model
is that it assumes the data are graph-structured, and that there is only a single input set, which
makes it difficult to compose together Pregel computations or combine two data sets.

In an attempt to find a more general parallel programming model, two projects have investi-
gated adding iteration support to MapReduce. Twister [ELZ+10] is a version of the MapRe-
duce execution engine that supports iterative computations by adding a “combine” stage that
runs after the reduce tasks, and computes a single value from the reduce results19. The re-
sult of the combine stage can then be used in a convergence test. Like Spark, Twister main-
tains an in-memory cache of any inputs that are re-used across iterations. HaLoop builds on

19Note that this is not related to the combiner optimisation in Google’s original MapReduce paper [DG04]. It
is also distinct from the merger tasks in Map-Reduce-Merge [YDHP07], because the combine stage computes a
single result from the outputs of the reduce tasks from a single MapReduce computation.
Hadoop MapReduce by adding a “loop control” component to the dependency resolver in the Hadoop master [BHBE10]. In HaLoop, an iterative MapReduce job can be specified as a normal MapReduce job with an additional ResultDistance function and a FixedPointThreshold value. The ResultDistance function computes the distance (in some metric space) between the outputs of two sets of reduce tasks. The HaLoop loop control continues to schedule MapReduce jobs until the distance between the two most recent job outputs becomes less than the FixedPointThreshold, or a predefined number of iterations is exceeded. HaLoop performs “loop-aware scheduling”, which aims to schedule tasks that process the same data on the same workers. The limitation of the iterative MapReduce model is that it does not support jobs that are more complicated than MapReduce (such as those computations that require a system supporting acyclic data-flow).

Research into data-flow programming (§2.2.3) has led to many architectures that support iteration by allowing cyclic data-flow. Beguelin et al. developed the Heterogeneous Network Computing Environment (HeNCE), which supports coarse-grained data-flow parallelism with explicit constructs for looping and conditional expressions [BDG+94]; however, HeNCE provides no fault tolerance for job execution. Davoli et al. developed the contemporary Paralex system, which is based on acyclic data-flow, with additional support for “cycle nodes” [DGB+96]. A Paralex cycle node contains an acyclic subgraph, which executes repeatedly, until the output of the subgraph makes the given “cycle control” function evaluate to true. This model places syntactic restrictions on iteration: all loops must have the same number of inputs as outputs, so that they may be fed back into the next iteration of a loop; furthermore, it does not support recursion, so all recursion must be translated into iteration. Paralex supports fault tolerance by replicating computation across process groups, which is inefficient if failures are rare.

### 2.3.5 Fork-join parallelism

In contrast to the above systems, systems based on dynamic task creation can easily support iterative algorithms because they have the facility to create more work based on the results of an intermediate computation. Conway introduced the fork-join model of parallelism, whereby a process can fork another process that runs in parallel, and later join that process to synchronise with it [Con63]. The Cilk programming language implements strict fork-join parallelism, which means that a function can spawn (fork) a thread to compute an expression but threads may only sync (join) with the thread that spawned them [BL99]. Blumofe and Lisiecki implemented Cilk-NOW, which was a distributed version of Cilk that runs on a network of workstations [BL97]. Cilk-NOW is well suited to a distributed environment, because it supports adaptive scaling to larger or smaller clusters, and it provides transparent fault tolerance. However, all data in a Cilk-NOW computation is passed by value in function parameters and return values, with no support for parallel I/O or distributed shared memory. Therefore, although it can compute simple functional expressions in parallel, Cilk-NOW is not suited to processing large amounts of data that cannot fit in RAM.
2.4 Summary

This chapter has surveyed a wide range of systems that support parallel computing, ranging from a single processor to a large data centre. The task-parallel systems discussed in Section 2.3 have become popular at the largest scale because their simple architecture is well suited to running on large clusters of potentially-unreliable components.

Among task-parallel systems, there has been a trend in recent years to increase the amount of coordination that the system performs on behalf of the programmer, which has greatly increased the class of algorithms that can run on these platforms, culminating in iterative systems that can support data-dependent control flow (§2.3.4). The main limitation of these iterative systems is that they do not support all of the algorithms that their less-expressive predecessors support: for example, the iterative MapReduce systems support iteration [BHBE10, ELZ+10], but not the full complement of acyclic data flows that Dryad supports [IBY+07]. Therefore there is a gap in the existing systems for a universal execution engine, which supports both data-dependent control flow and a superset of the existing systems’ programming models. In the remaining chapters of this dissertation, I will discuss CIEL, which is the first system to meet this objective.
Chapter 3

Dynamic task graphs

The previous chapter featured many examples of systems that use coarse-grained data flow to achieve massive parallelism. Distributed execution engines—in particular MapReduce [DG04] and Dryad [IBY+07]—have enhanced the popularity of data-flow parallelism, because they expose the abstraction of a reliable machine for executing sequential code, which can then be executed in parallel on a large cluster of commodity machines. However, these frameworks are limited because they use static acyclic data-flow graphs, and hence they cannot express unbounded iterative or recursive algorithms. More recently, systems such as Twister [ELZ+10], Pregel [MAB+10] and Piccolo [PL10] have made progress in supporting iterative computations, but the programming model for each iteration is less expressive than a Dryad DAG. Furthermore, these systems rely on stateful processing nodes: hence they do not support transparent fault tolerance, and must use more heavyweight solutions (e.g. checkpointing).

This chapter introduces the dynamic task graph execution model, which is an extension of acyclic data flow that supports data-dependent control flow, and hence is Turing-complete. The computational power in a dynamic task graph arises from the ability of a task to spawn additional tasks: for example, a task can update the graph to include an additional iteration.

Dynamic task graphs are very general: in defining the model, I make few assumptions about the behaviour of an individual task, or the structure of its inputs or outputs. Obviously, since each task can be written in a Turing-complete language, it would be possible to execute any algorithm by performing all data-dependent control flow within a single task (or finite set of tasks). However, it is possible to restrict task execution to a bounded duration, without compromising the expressiveness of dynamic task graphs. Hence the model is practical: it does not shift the burden of data-dependent control flow onto a single task.

In Section 3.1, I begin by formally defining the key components of a dynamic task graph. In Section 3.2, I characterise dynamic task graphs as a form of labelled transition system, and present an algorithm for identifying tasks to execute. Finally, in Section 3.3, I demonstrate the generality of the model by showing how other models of computation can be reduced to dynamic task graphs, and showing that the dynamic task graph model is Turing-complete.
3.1 Definitions

This section introduces the principal components of a dynamic task graph in bottom-up order. Each concept is introduced with a formal definition, an informal explanation and—where appropriate—a graphical representation that will be used in the remainder of this dissertation.

**Definition 1 (Concrete object)** A concrete object, \( o \in \text{Obj} \), is an immutable, unstructured and finite-length sequence of bytes. Without loss of generality, let \( \text{Obj} = \mathbb{N} \).

Concrete objects are used to represent input, intermediate and output data in a dynamic task graph. Figure 3.1(a) shows the symbol for a concrete object. Note that the definition of a concrete object is flexible: for example—unlike MapReduce [DG04], Hadoop [Had] and Dryad [IBY+07]—it is not necessary to structure data as a sequence of records. This allows the use of objects to store the code for tasks, as discussed below.

**Definition 2 (Future object)** The future object, \( \varphi \notin \text{Obj} \), corresponds to any object that has not yet been produced. The future object is an object that cannot be read, and is used as a placeholder for objects that have not yet been produced. For example, when a task is first created, all of its outputs are represented by the future object. Figure 3.1(b) shows the symbol for the future object.

The future object is an object that cannot be read, and is used as a placeholder for objects that have not yet been produced. For example, when a task is first created, all of its outputs are represented by the future object. Figure 3.1(b) shows the symbol for the future object.

**Definition 3 (Names and Stores)** A name, \( n \in \text{Name} \), is an opaque identifier that may be used to refer to an object. A name is itself an object, so \( \text{Name} \subseteq \text{Obj} \).

A store is a mapping, \( \Sigma \in \text{Store} = \text{Name} \times (\text{Obj} \cup \{ \varphi \}) \), which maps a set of names to unique concrete objects or the future object. Formally, \( \forall n \in \text{Name}, \forall \Sigma_i, \Sigma_j \in \text{Store}, (o_k, o_l \in \text{Obj} \land \Sigma_i(n) = o_k \land \Sigma_j(n) = o_l) \Rightarrow o_k = o_l \).
The combine operation, $\oplus : \text{Store} \times \text{Store} \rightarrow \text{Store}$, may be used to produce a new store from two stores. If $\Sigma_i \oplus \Sigma_j = \Sigma_k$, then $\Sigma_k$ can be defined as follows:

$$
\Sigma_k(n) = \begin{cases} 
\text{undefined} & \text{if } n \notin \text{dom } \Sigma_i \cup \text{dom } \Sigma_j \\
o_i & \text{if } \Sigma_i(n) = o_i \land o_i \in \text{Obj} \\
o_j & \text{if } \Sigma_j(n) = o_j \land o_j \in \text{Obj} \\
\varphi & \text{otherwise}
\end{cases}
$$

Every object in a dynamic task graph has a name, which is bound in one or more stores. Figure 3.2 shows an example store, containing three name-to-object mappings. The store plays three roles in a dynamic task graph. First, it enables large concrete objects to be passed by reference, which allows the graph to describe computations that process large data without having to store all of the data in the graph. Secondly, since a name may only map to one object, this implies that references are immutable, which enables an object to be replicated for performance or reliability [GGL03]. Finally, the ability of the store to map names to the future object enables scheduling constraints to be expressed in terms of names: if a task depends on a name that currently maps to the future object, that task cannot be scheduled until a store exists with a mapping from that name to a concrete object.

The combine operation is used to allow the “current” store to evolve as a job executes; it is used in the transition system that I will define in §3.2.1. Informally, combining two stores produces the union of the two stores, except that, when both stores contain a mapping for the same name, the new store will always prefer a concrete object, if one exists. The uniqueness constraint ensures that, if $\Sigma(n)$ maps to an object $o$ in any store, it will map to $o$ in all stores, so the operation is commutative. Note that the universe of stores (and hence the namespace) may be defined for a single job or globally for all jobs; the actual implementation of naming (§4.2.1) uses a global store in order to allow memoisation between different jobs.
CHAPTER 3. DYNAMIC TASK GRAPHS

Figure 3.3: A task, \( t = \langle d_{\text{code}}; \{d_1, d_2\} \rangle \leadsto \{e_1, e_2, e_3\} \). (a) The task is shown in the context of a store in which \( d_{\text{code}} \) and \( d_2 \) refer to concrete objects, and \( d_1 \) refers to the future object. (b) When the code dependency is concrete, the diagram may be abbreviated by labelling the task itself with the name of the code dependency.

**Definition 4 (Task)** A task, \( t \in \text{Task} \), is defined in terms of its dependencies: \( t = \langle d_{\text{code}}; \{d_1, \ldots, d_m\} \rangle \), where \( d_{\text{code}} \in \text{Name} \) is the code dependency, and \( d_1, \ldots, d_m \in \text{Name} \) are the data dependencies. Let \( D_t \) denote the dependency set of task \( t \), \( D_t = \{d_{\text{code}}, d_1, \ldots, d_m\} \). A task also has a set of expected outputs, \( E_t = \{e_1, \ldots, e_n\} \subseteq \text{Name} \), which can be computed from the dependencies. However, for convenience, a task may be written with explicit expected outputs as \( t = \langle d_{\text{code}}; \{d_1, \ldots, d_m\} \rangle \leadsto \{e_1, \ldots, e_n\} \).

A task is **well-defined** with respect to a store \( \Sigma \), if and only if \( \forall d \in D_t, d \in \text{dom} \Sigma \).

A task is **runnable** with respect to a store \( \Sigma \), if and only if \( \forall d \in D_t, \Sigma(d) \in \text{Obj} \).

Tasks are the atomic unit of computation in a dynamic task graph. The dependency set names the objects that are inputs to the task, and the set of expected outputs names the objects that the task is expected to produce. The code dependency is a special dependency that denotes the task **behaviour**: note that this simply names an object, which means that the task behaviour may be provided as an input, or it can be produced by another task. The code dependency need not represent the complete behaviour of a task: for example, Chapter 5 introduces Java tasks, which have one or more JAR files as data dependencies, and a code dependency that includes the names of the objects containing those JAR files. The ability to fulfil code dependencies dynamically is useful when expressing data-dependent control flow, as it facilitates dynamic task spawning (see Definition 6).
CHAPTER 3. DYNAMIC TASK GRAPHS

Figure 3.3 shows a task with three dependencies, and three explicit expected outputs, which are depicted as future objects. These are “expected” since a task may not in fact produce all of its expected outputs, but instead dynamically delegate their production to one or more dynamically spawned tasks, as discussed in Definition 7. It is not strictly necessary for a task to declare its expected outputs in the task graph, because they can be computed from the task dependencies. However, doing so enables an implementation to place constraints on the dynamic task graph that ensure progress, as discussed in Section 3.2.

A task is runnable if and only if all of its dependencies can be resolved to concrete objects in the current store. This definition of runnability facilitates deterministic synchronisation between tasks, even though each dependency may be resolved by a different task running in parallel, and hence in a non-deterministic order. Determinism is a conservative requirement, which is discussed below in the context of executors (Definition 6) and later in the context of fault tolerance (Section 4.4). Other definitions of runnability are possible: for example, one could imagine a task becoming runnable when any one of its dependencies can be resolved to a concrete object, which would permit non-deterministic synchronisation; I will explore this possibility in Chapter 7.

**Definition 5 (Task graph)** A task graph, \( \Gamma = (\Sigma, T) \), where \( \Sigma \in \text{Store} \) and \( T \subset \text{Task} \) represents the current state of a computation.

A task graph is a static snapshot of a dynamic task graph; I will discuss how task graphs evolve dynamically in §3.2. A task graph comprises a set of tasks, and a store containing the names of objects on which those tasks depend. As Figure 3.3 shows, the object names and tasks comprise the vertices in a task graph, and the edges are derived from the dependency and expected output sets of each task. The graph is bipartite: an edge from name \( n \) to task \( t \) implies that \( t \) depends on the object named \( n \); an edge from \( t \) to \( n \) implies that \( t \) is expected to produce an object named \( n \); there are no edges between two names or between two tasks.

In practice, not all task graphs are valid. Only acyclic task graphs are considered in this dissertation, and the invariants described in §3.2 ensure that all task graphs are acyclic. A cyclic task graph could be conceived, but it would complicate the implementation of fault tolerance, because it would be difficult to identify the tasks to re-execute in the event of a failure. Furthermore, cyclic dependencies could lead to deadlock, which would complicate the scheduler. Since a dynamic task graph can perform iteration, and the scheduling optimisations described in Chapter 4 can reduce the cost of repeatedly processing the same input, there is no need to support cycles in the execution model. By only allowing acyclic task graphs, it is possible to use a variation of existing techniques, which use topological ordering, to evaluate the static portions of a dynamic task graph (§3.2.2).
**Definition 6 (Executor)** The executor is a function, \( E : \text{Task} \rightarrow \text{Store} \times \mathcal{P}(\text{Task}) \). Given a task \( t \), the result of evaluating \( E(t) \) is a task graph, \( \langle \Sigma_t, T_t \rangle \).

If \( \Sigma_t(n) = o \) and \( o \in \text{Obj} \), \( n \) is said to be *produced* by \( t \).

The tasks in \( T_t \) are said to be *spawned by* or *children of* \( t \).

The executor is an abstraction of the actual machine or (distributed) system that executes a computation. In general, it is a component that is responsible for interpreting the semantics of a task’s code and data dependencies. Applying the executor to a task (hereafter “executing a task”) produces a structure that resembles a task graph: \( \langle \Sigma_t, T_t \rangle \) is in fact the subgraph that is added to the dynamic task graph when \( t \) is executed (§3.2.1). In Chapter 5, I will present concrete examples of executors that can support various programming models on top of dynamic task graphs; however, in the interests of generality, this chapter and Chapter 4 do not assume the use of any particular executor.

There are various pragmatic constraints on the implementation of an executor to make it suitable for distributed data-flow programming. The most important constraint is that executing a task \( t \) must depend only on objects that are explicitly named in the dependency set of \( t \), viz. \( D_t \). This ensures that all data-flow is explicit in the dynamic task graph, and task execution may proceed independently of the production of any object not named in \( D_t \); hence independent tasks can execute correctly in parallel. A secondary constraint is that \( E \) should be a *deterministic* function: i.e. \( \forall t \), evaluating \( E(t) \) must always produce the same result; this allows fault tolerance through re-execution, as discussed in Section 4.4. Finally, the implementation of \( E \) should be *local* and *non-blocking*: i.e. for any \( t \), evaluating \( E(t) \) should always run to completion using the computational resources of a single processing element. If a task were allowed to block on its spawned tasks while occupying a processor, this could lead to deadlock if insufficient processors were available to run the child (and its descendants).

It would be desirable to make the executor a *total function*: i.e. to guarantee termination for all tasks. It would even more desirable to restrict task execution so as to bound the amount of time spent executing each task. Many of the example algorithms that run on MapReduce and Dryad—including word counting, regular expression matching and sorting [DG04], and relational query processing and histogramming [IBY^+07]—are guaranteed to terminate, and perform a bounded amount of work for each input record. However, in practice, these systems allow programmers to define their own functions in general-purpose Turing-powerful languages—such as C++ [DG04], Java [CRP^+10, Had] and C\(^\#\) [CJL^+08, YIF^+08]—for which termination cannot in general be proven, due to the halting problem [Tur36]. Therefore, to support existing programming models, dynamic task graphs should not *enforce* the requirement of guaranteed termination on task execution. However, as I will show in §3.3.3, restricting tasks to bounded duration does not limit the expressiveness of the dynamic task graph model.
Definition 7 (Task dependencies and delegation) The task dependency relation, $\prec$, is a partial order on the tasks in a task graph, $\Gamma = \langle \Sigma, T \rangle$. Task $u$ depends on task $t$ (i.e. $t \prec u$) if one or more of the following conditions hold:

Child dependency $E(t) = \langle \Sigma_t, T_t \rangle$ and $u \in T_t$.

Data dependency $E(t) = \langle \Sigma_t, T_t \rangle$ and $D_u \cap \text{dom } \Sigma_t \neq \emptyset$.

Indirect dependency $\exists v \in T$ such that $t \prec v$ and $v \prec u$.

For tasks $t, u$, where $t \prec u$ and $E_t \cap E_u \neq \emptyset$, the expected outputs in $E_t \cap E_u$ are said to be delegated from $t$ to $u$.

Delegation is said to be strict when $\forall t, u \in \text{Task}, E_t \cap E_u \neq \emptyset \Rightarrow t \preceq u \lor u \preceq t$.

A task $t_{\text{max}}$ is said to be maximal in a set of tasks, $T$, if $\forall t \in T$, $t_{\text{max}} \preceq t \Rightarrow t_{\text{max}} = t$. A minimal task can be defined dually.

Let $n \in \text{Name}$ and $P_{n,T} = \{t \mid t \in T \land n \in E_t\}$ be the expected producer set of $n$. A maximal task in $P_{n,T}$ is said to be a maximal expected producer of $n$. If delegation is strict, there is a unique maximal expected producer for each $n$.

The task dependency relation extends the notion of dependencies to an entire task graph: if the relation is not defined for a pair of tasks, those tasks can execute in parallel; otherwise, $t \prec u$ implies that $t$ must execute before $u$. This is a well-known property of graphical models of computation [KM66], and is the underlying principle in Dryad [IBY07] and similar systems.

Delegation is an important concept that supports data-dependent control flow in a dynamic task graph. Normally, when a task spawns a child task, fresh names are created for each of the child task’s expected outputs, and each name is initially mapped to the future object. In order for the results of a spawned task to be used for data-dependent control flow, they must be passed as dependencies to one or more new tasks; however, spawning a new task will create a new name, which poses the same problem, and so on ad infinitum. Since blocking is not permitted during task execution (for reasons discussed above), the parent task must terminate before its children terminate. Instead of producing its outputs, the parent task may delegate the production of its output to a child, by spawning a child task with the same expected outputs as its parent. Delegation is similar to programming in a tail-recursive style [Ste77]: a regular spawn is analogous to a stack-based call, whereas a delegating spawn is analogous to a tail call. Under strict delegation, a task may only delegate each of its outputs once, to preserve the total order between expected producers. This does not limit expressive power, and admits a safer programming model, based on the tail-spawn operation (§5.2.2). Therefore, in the remainder
of this dissertation, all delegation is assumed to be strict. Figure 3.4 shows an example of strict delegation in action: a task spawns two child tasks, and delegates its output to a third child task.

Delegation creates a bookkeeping challenge: since many different tasks may be expected to produce a given output, which is the correct task to execute? Intuitively, the “most recently created” task is likely to be correct, since in normal execution the delegating spawns will occur in the correct order. However, in the event of failures (see Section 4.4 for more details), some tasks may have to be re-executed in order to make progress. The task dependency relation solves this problem: the correct task to execute in order to produce a given output is the maximal expected producer for that output, as defined above.

**Definition 8 (Job)** A job is a task graph, $\Gamma_{\text{root}} = (\Sigma_{\text{root}}, \{t_{\text{root}}\})$, where $\Sigma_{\text{root}}$ is the initial store, $t_{\text{root}} \in \text{Task}$ is the root task and $E_{t_{\text{root}}}$ is the set of job output names.

The initial store, $\Sigma_{\text{root}}$, must contain mappings such that $\forall d \in D_{t_{\text{root}}}, \Sigma_{\text{root}}(d) \in \text{Obj}$.

A job terminates on a store, $\Sigma$, if $\forall e \in E_{t_{\text{root}}}, \Sigma(e) \in \text{Obj}$.

Intuitively, a job is the base case of a dynamic task graph, which comprises a single root task and a store that contains sufficient concrete objects to make the root task runnable. The aim of running a job is therefore to produce an object for each of the root task’s expected outputs. In
order to achieve this, the root task may spawn child tasks and delegate production of the job outputs to its children.

This definition of a job is quite different from existing frameworks. In fixed data-flow frameworks (§2.3.2), a job is specified using a simple configuration (e.g. a key-value dictionary) that specifies how many of each kind of task should be created, and the names of the inputs. In acyclic data-flow frameworks (§2.3.3), the explicit set of task dependencies must be specified, for example using a domain-specific language for building directed acyclic graphs. By contrast, a dynamic task graph is constructed by executing the root task and any subsequent tasks. The following section discusses this process in more detail.

3.2 Executing a dynamic task graph

The previous section defined the various components and properties of a dynamic task graph in terms of static task graphs, which are snapshots of the overall computation. This section explains how a dynamic task graph evolves as a side effect of executing tasks. Subsection 3.2.1 presents the execution relation between two task graphs at points before and after a task executes, and introduces consistency properties to ensure that the computation always makes progress. Subsection 3.2.2 then introduces a dependency resolution algorithm for dynamic task graphs, which is based on lazy evaluation.

3.2.1 Graph dynamics

A dynamic task graph is a labelled transition system in which the states are task graphs (Definition 5), transitions are defined for runnable tasks (Definition 4) in a given state, and the successor state is obtained by combining the current task graph with the results of applying the executor function (Definition 6) to a runnable task.

States are related by the state transition relation, $\xrightarrow{t}$, which is defined as follows:

\[
\begin{align*}
    t &\in T \quad \forall d \in D_t, \Sigma(d) \in \text{Obj} \\
    \langle \Sigma, T \rangle &\xrightarrow{t} \langle \Sigma \oplus \Sigma_t, T \cup T_t \rangle
\end{align*}
\]

or, in words:

There is a runnable task that spawns a set of tasks and produces a set of objects

The current task graph can be updated to contain the new tasks and objects

The $\xrightarrow{t}$ relation connects two task graphs, but it does not prescribe a strategy for selecting the task $t$ that should be executed. However, the definition has three useful properties. First, note that task execution is idempotent, because the side effects of executing a task are limited to idempotent operations on the store and task set ($\oplus$ and $\cup$, respectively): this allows tasks to
be re-executed (e.g. in the event of a network partition) without additional coordination logic. Second, task execution is **commutative**: if there are two runnable tasks \( t \) and \( u \) in a particular graph \( \Gamma_\alpha \), then \( \Gamma_\alpha \xrightarrow{t} \Gamma_\tau \xrightarrow{u} \Gamma_\omega \) and \( \Gamma_\alpha \xrightarrow{u} \Gamma_\upsilon \xrightarrow{t} \Gamma_\omega \) (for some intermediate graphs \( \Gamma_\tau, \Gamma_\upsilon \)). This enables flexible parallelism in a system implementing dynamic task graphs, since tasks can complete in a non-deterministic order without affecting the correctness of the result. Finally, task execution is **monotonic**: after executing a task, the store and task set are superset of their previous values, which means that all previous tasks remain well-defined and runnable. This allows an implementation of fault tolerance based on task re-execution, as I will describe in Section 4.4.

In addition to the execution relation, it is helpful to define easily-checkable consistency properties that ensure that a computation continues to make progress. Note that it is impossible to guarantee that an arbitrary dynamic task graph always terminates, because the model is Turing-complete and hence can describe infinite computations. However, there are several inconsistent task graphs that neither terminate nor make progress, and it would be desirable to eliminate these programmatically.

**Output fulfilment** For each expected output of a task, that task must either: (i) produce that output or (ii) spawn a new task to which the production of that output is delegated. Formally:

\[
t \in T \land e \in E_t \land E(t) = (\Sigma_t, T_t) \Rightarrow \Sigma_t(e) \in \text{Obj} \lor \exists u \in T_t \text{ s.t. } e \in E_u
\]

If a task has an expected output, a subsequent task may depend on that expected output. However, if the first task neither produces the expected output nor delegates its creation to a child task, the dependent task will never become runnable. Therefore, the task graph would be inconsistent.

**Data dependency** When a task is created, its dependency set may only contain names that are known to the store, either as input data (defined in \( \Sigma_{\text{root}} \)) or as the output of an existing task. Formally:

\[
t \in T \land d \in D_t \Rightarrow \Sigma_{\text{root}}(d) \in \text{Obj} \lor (\exists u \in T \text{ s.t. } d \in E_u \land u \prec t)
\]

This property ensures that neither cyclic nor dangling data dependencies can form in the dynamic task graph.

In practice it is not always necessary to check these properties, because the programming model can ensure that they cannot be broken. Chapter 5 introduces various programming models that ensure that these properties always hold.

---

[1] Here, monotonicity can be defined by analogy with a monotonic Datalog program, which only ever adds tuples to a relation [KV90]. Executing a task only ever adds tasks and bindings to the dynamic task graph.
3.2.2 Evaluation strategy

So far, this section has discussed how the task graph changes as tasks are executed, but it has not specified how task dependencies are resolved, nor how tasks are chosen for execution. According to Definition 4, a task is runnable if and only if all of its dependencies are concrete. Unlike the fork-join model, in which a forked task is immediately runnable and continuations form a stack, dynamic task graphs permit arbitrary acyclic dependencies between spawned tasks, so some form of dependency resolution is necessary [BGM99]. In a static acyclic data-flow graph, the order of execution can be determined by topologically sorting the graph [Las61]; furthermore, the channel between producing and consuming tasks is direct, so the data motion is also pre-determined. Unlike the static acyclic data-flow model, the challenge in a dynamic task graph is that the full graph is not known in advance. This subsection discusses an evaluation strategy that can be used to determine the next task to execute.

When a job is submitted, it contains only one task—the root task—which by definition has no unfulfilled dependencies, so the only way to make progress is to execute that task. When the root task has executed, it will either have spawned a new set of tasks, or produced its output. Assume that it spawns a new set of tasks; then, according to the data dependency invariant in §3.2.1, that set of tasks must form an acyclic graph, so there must be at least one task with no unfulfilled dependencies.

The simplest evaluation strategy is eager evaluation, which is based on Kahn’s algorithm for topological sorting [Kah62]. In this strategy, whenever the task graph contains a task with no unfulfilled dependencies, that task is added to the run queue. To achieve this, when a task is spawned, it subscribes to each of its unfulfilled dependencies, and maintains a count of unfulfilled dependencies. When a dependency becomes concrete in the current store, all of the tasks subscribed to the corresponding name are notified, by decrementing their counts of unfulfilled dependencies; if a task’s count becomes zero, that task is added to the run queue. Eventually, if the algorithm of the job converges, some task will execute that produces the job output, and hence terminates the job.

The advantage of the eager strategy is its simplicity: there is no need to include explicit logic for delegation; the implicit temporal dependency between parent and child tasks ensures that delegated tasks execute in the correct order. The main disadvantage is that the approach does not provide efficient fault tolerance. As discussed further in Subsection 4.4.2, in a data-flow system, a task may fail because one of its dependencies is no longer available: in this case, some tasks must be re-executed in order to reproduce the object of the dependency. Eager evaluation proceeds by making forward progress from the root task to the eventual result, whereas an efficient fault recovery mechanism would start with the missing object and work backwards, only re-executing the tasks that are necessary to reproduce the object.

---

2 This algorithm considers all the vertices having no incoming edges, removes their outgoing edges, then adds those vertices to the sorted list. It then repeats by considering all the vertices that now have no incoming edges.
\( R \leftarrow \emptyset, B \leftarrow \emptyset, Q \leftarrow \text{empty queue} \)

\[
\text{expected producer} \leftarrow \max\{t \mid t \in T \land o \in E_t\}
\]

\textbf{append} expected producer to \( Q \)

\textbf{while} \( Q \) is not empty \textbf{do}

\hspace{1em} current task \leftarrow \textbf{remove first} from \( Q \)

\hspace{2em} \textbf{if} current task is runnable \textbf{then}

\hspace{3em} \( R \leftarrow R \cup \{\text{current task}\} \)

\hspace{2em} \textbf{else}

\hspace{3em} \( B \leftarrow B \cup \{\text{current task}\} \)

\hspace{2em} \textbf{for all} \( d \in D_{\text{current task}} \) \textbf{do}

\hspace{3em} \textbf{if} \( \Sigma(d) = \emptyset \) \textbf{then}

\hspace{4em} \text{expected producer} \leftarrow \max\{t \mid t \in T \land d \in E_t\}

\hspace{3em} \textbf{if} expected producer \not\in R \cup B \cup Q \textbf{then}

\hspace{4em} \textbf{append} expected producer to \( Q \)

\textbf{Figure 3.5:} Lazy evaluation algorithm for evaluating object \( o \) in a dynamic task graph.

After running the algorithm, \( R \) is the set of runnable tasks, and \( B \) is the set of blocked tasks.

Since fault tolerance is an important feature in a distributed execution engine, I have devised a strategy that resembles Henderson and Morris’s “lazy evaluator” [HM76] and hence is called lazy evaluation. Informally, this approach results in the executions of the sub-graph of tasks that is necessary and sufficient to produce a target object, \( o \). The resulting algorithm (Figure 3.5) is somewhat more complicated than the eager evaluation algorithm: it performs a breadth-first search through the data-flow graph to identify the set of tasks that contribute to the production of \( o \), and partitions them into a runnable set, \( R \), and a blocked set \( B \). The tasks in \( R \) are then added to the run queue, while the tasks in \( B \) subscribe to their dependencies as in the eager evaluation algorithm. It would be possible—but inefficient—to execute this algorithm for the job output each time the dynamic task graph is updated. However, a more efficient approach is to retain the sets \( R \) and \( B \) between invocations of the algorithm. To avoid traversing the entire graph, the algorithm is then executed for the set of names that were delegated by the most recently-completed task.

Just as lazy evaluation in functional programs is related to \textit{call-by-need} semantics [HM76], lazy evaluation in a dynamic task graph only executes a task if its execution is necessary to produce a job output, either by producing that output or some object on which a blocked task depends. As a result, it does not execute a task if its outputs are already concrete. Therefore lazy evaluation naturally supports memoisation (§4.2.1) and fault tolerance by recursive task re-execution (§4.4.2).
3.3 Relation to other execution models

The aim of this chapter is to introduce dynamic task graphs as a universal execution model for parallel computing. In this context, universality has two separate, but related, meanings:

**Applicability to existing models** There must exist an automatic and efficient translation from existing execution models (including MapReduce [DG04], Dryad [IBY+07] and Bulk Synchronous Parallel [Val90]) to a universal execution model.

**Effective calculability** A universal execution model must be able to compute all effectively-calculable functions [Chu36]. As Rosser notes [Ros39], this is equivalent to the set of problems that can be solved by a Turing machine, so a universal execution model must be Turing-complete.

Furthermore, it is imperative that the universal execution model be practical. For example—in the context of dynamic task graphs—it would be unsatisfying to remark that each task is Turing-powerful, and therefore the model is universal. It is undesirable for the computation in a single task to be unbounded: since tasks may be opaque functions, it is only at the boundaries of a task that the system (Chapter 4) can provide distributed coordination, communication between tasks and fault tolerance. Therefore, encapsulating unbounded computation in a single task limits the parallelism that can be achieved, and increases the exposure to faults.

Therefore, in this section, I will restrict the executor function, $E$, to be a total function: i.e. it is defined $\forall t \in \text{Task}$. This is a practical restriction: the execution of a single task must terminate on all valid inputs, and report an error within bounded time on all invalid inputs. In practice, tasks in a distributed execution engine are assumed to terminate [KSV10], but implemented in a Turing-complete language, which makes it intractable to prove this statically for all inputs.

The general approach in this section is to posit the existence of a variety of objects that, when specified as the code dependency of a task, cause the executor ($E$) to produce a specified result. This can be justified by noting that the code dependency can be an arbitrary piece of machine code, which a concrete executor will execute (see Chapter 5 for several examples of this). However, to ensure that $E$ is a total function, I will restrict the code dependency to specify only computations that are guaranteed to terminate.

Consider an algorithm for computing, in parallel, the $n$th Fibonacci number, $F_n = F_{n-1} + F_{n-2}$. The initial job graph ($\Gamma_{\text{root}}$) could contain three objects—one representing the input ($n$), and two representing the code dependencies of all subsequent tasks ($t_F, t_+$)—and one task, which is expected to compute $F_i$. A $t_F$ task would compute the $i$th Fibonacci number, where $i$ is passed as the sole data dependency, and its behaviour is data-dependent. If $i > 1$, the task would spawn two $t_F$ tasks to produce $F_{i-1}$ and $F_{i-2}$; and a $t_+$ task that would depend on $F_{i-1}$ and $F_{i-2}$, and produce the original task’s output ($F_i$). Otherwise, if $i$ is 0 or 1, the task would simply write the input value as its output. A $t_+$ task, by contrast, is simpler: it would unconditionally output the
Table 3.1: Definition of a dynamic task graph for recursively calculating the $n^{th}$ Fibonacci number.

<table>
<thead>
<tr>
<th>$t$</th>
<th>$\Sigma$</th>
<th>$T$</th>
<th>condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Initial job)</td>
<td>$n \mapsto \text{input}$</td>
<td>$\langle t_F; {n} \rangle \mapsto {F_n}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$t_F \mapsto \text{see below}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\langle t_F; {d_i} \rangle \mapsto {F_i}$</td>
<td>$d_{i-1} \mapsto \Sigma(d_i) - 1$</td>
<td>$\langle t_F; {d_{i-1}} \rangle \mapsto {F_{i-1}}$</td>
<td>$\Sigma(d_i) &gt; 1$</td>
</tr>
<tr>
<td></td>
<td>$d_{i-2} \mapsto \Sigma(d_i) - 2$</td>
<td>$\langle t_F; {d_{i-2}} \rangle \mapsto {F_{i-2}}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$d_i \mapsto \Sigma(d_i)$</td>
<td>$\emptyset$</td>
<td>$\Sigma(d_i) \in {0, 1}$</td>
</tr>
<tr>
<td>$\langle t_+; {d_x, d_y} \rangle \mapsto {z}$</td>
<td>$z \mapsto \Sigma(d_x) + \Sigma(d_y)$</td>
<td>$\emptyset$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.6: Dynamic task graphs for recursively calculating the $n^{th}$ Fibonacci number.

sum of its two data dependencies. Table 3.1 summarises this description in tabular form, and Figure 3.6 shows the dynamic task graphs that would result.

In the remainder of this section, I will show how dynamic task graphs can be used to represent three existing execution models: MapReduce (§3.3.1), Bulk Synchronous Parallel (§3.3.2) and while programs (§3.3.3). Since while programs are Turing-complete [BJ66], this amounts to proving the Turing-completeness of dynamic task graphs.

### 3.3.1 MapReduce

Dean and Ghemawat’s MapReduce [DG04] is a fixed data-flow programming model, which comprises two phases: a map phase of $m$ tasks, followed by a reduce phase of $r$ tasks. (MapRe-
Table 3.2: Definition of a dynamic task graph for performing a MapReduce computation. The MAP($i_k$, 1) function returns the $j^{th}$ partition of applying the mapper to input $i_k$. The RED($m_{1,k}$, $m_{m,k}$) function returns the output of applying the reducer to the $k^{th}$ intermediate outputs from each mapper.

reduce is introduced more fully in §2.3.2.) Each task in the map phase writes $r$ outputs, and each reduce task reads $m$ inputs: the $i^{th}$ reduce task reads the $i^{th}$ output from each map task. There is no data-dependent control flow at the job level, so each task runs to completion and produces all of its outputs. In this subsection, I assume that the user provides map and reduce functions that are guaranteed to terminate.

The basic strategy in translating a MapReduce job into a dynamic task graph uses a root task to spawn both the map and reduce phases. Table 3.2 shows the complete definition of a dynamic task graph for computing a MapReduce job. The initial graph contains $m$ objects representing the “input splits” (partitions of the input file, in Dean and Ghemawat’s model), and various objects to represent the tasks, including objects representing the map function ($t_M$) and the reduce function ($t_R$). The root task, $t_{root}$, spawns $m + r + 1$ tasks: $m$ map tasks, $r$ reduce tasks and one further task for synchronisation.

The $t_M$ task depends on a single input split, and produces $r$ outputs: the outputs contain the result of executing the map function on each input record. The result of the map function, applied
to a record, is zero or more key-value pairs, and these are partitioned by key between the outputs according to a user-defined function, and sorted by key within each partition. In Table 3.2, this is represented by the MAP function. Note that the MapReduce framework includes code—for parsing input data, and sorting and partitioning intermediate data—that executes in map tasks, and this characterisation assumes that this library code is statically linked with $t_M$. The $t_R$ task depends on $m$ intermediate objects, and produces a single output containing the result of executing the reduce function on each unique key in the combined input data and the bag of values associated with that key. Again, this assumes that library code for merging the sorted intermediate data is statically linked with $t_R$. Finally, the $t_{sync}$ task depends on all $r$ reduce outputs, and has the effect of concatenating the names—but not the data—of those outputs in a single list, which is the overall job output (delegated from the root task).

Figure 3.7 illustrates a MapReduce dynamic task graph, with three map tasks and two reduce tasks. Note that the numbers of map and reduce tasks are the key parameters of a MapReduce job: the number of map tasks can be inferred from the number of input splits, whereas the above translation assumes that the number of reduce tasks is encoded in $t_M$. 
Although I have only discussed MapReduce in this subsection, the approach of having a root task that spawns a static task graph is suitable for implementing any data-flow model that can be represented by a static DAG. Therefore, the wavefront [YME+09] and all-pairs [MBH+10] fixed data-flow models can also easily be simulated, while Dryad [IBY+07] can be supported by passing a representation of the data-flow DAG as an input to the root task.

### 3.3.2 Bulk Synchronous Parallel

Valiant proposed the Bulk Synchronous Parallel (BSP) execution model as a universal bridging model from the shared memory model to message-passing architectures. (BSP is introduced in §2.2.2.) A BSP computation is organised into a sequence of supersteps. During a superstep each processor executes in parallel, then all processors exchange messages with one another; the number of supersteps is unbounded. This subsection shows how a BSP computation can be implemented using a dynamic task graph. The Pregel variant of BSP, introduced by Malewicz et al. [MAB+10], is used for concreteness and because it incorporates a notion of termination that is lacking in the original definition of BSP. In each superstep of a Pregel computation, the workers apply a vertex function to the partitioned vertices of a large graph in parallel, and each vertex may vote for the computation to halt; the computation terminates when all vertices have voted to halt.

The translation from a Pregel computation to a dynamic task graph uses the root task to spawn the first superstep; and a master task to collect votes and—depending on the votes—spawn further supersteps or produce a result. Table 3.3 shows the complete definition of a dynamic task graph for performing a Pregel computation. The initial dynamic task graph contains $n$ objects representing the initial partitions of the graph data (hereafter just “partitions” to avoid confusion with the dynamic task graph). In the table, the notation $i_k$ is used to denote the $k^{th}$ input partition ($1 \leq k \leq n$). The initial graph also contains a vertex program, named $t_V$, which encodes the operation carried out on every element in the data set during each superstep. The root task, $t_{\text{root}}$, spawns $n + 1$ tasks: $n$ vertex tasks, and one master task, which depends on outputs from all of the vertex tasks.

In the first superstep, a vertex ($t_V$) task depends on a single input partition, $i_k$, and produces $n + 1$ outputs, comprising:

**Message objects** A vertex task produces $n$ objects containing the messages that will be sent to the $n$ partitions before the next superstep ($m^1_{k,1}, \ldots, m^1_{k,n}$). To simplify the notation, the message-to-self, $m^1_{k,k}$, also includes the updated mutable state of the $k^{th}$ partition. As in Pregel, a combiner may be applied to aggregate the messages.

**Vote object** A vertex task also produces a single vote object, $v^1_k$, containing a Boolean value. In Pregel, each element in the data set may vote to halt; the algorithm terminates when the entire data set unanimously votes to halt. The vote object is true if and only if all elements in the $k^{th}$ partition vote to halt.
Table 3.3: Definition of a dynamic task graph for performing a Pregel (Bulk Synchronous Parallel) computation

The outputs are produced by iterating over the vertices in a partition, as described by Malewicz et al. [MAB+10]: each vertex may update a private mutable value and send messages to any other vertex. A user-defined partitioning function distributes messages to the appropriate message object, and hence to the appropriate partition in the next superstep. In subsequent supersteps, a $t_V$ task additionally depends on messages from other partitions, which are made available to each vertex via its incoming message iterator.

In the $i$th superstep, the master ($t_M$) task depends on the vote objects ($v^i_k$) produced by each vertex task. The master task aggregates the votes, and uses the result to determine the appropriate action. If all partitions (and hence elements) vote to halt, the master task publishes the job output: a list of the names of the current partitions. Otherwise, at least one vertex has voted to continue the computation, so the master task spawns another superstep, with $n$ vertex tasks, and another master task to which the job output is delegated.

Figure 3.8 shows the state of a dynamic task graph during the second superstep of a Pregel

<table>
<thead>
<tr>
<th>$t$</th>
<th>$\Sigma_t$</th>
<th>$T_t$</th>
<th>condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Initial job)</td>
<td>$i_1, \ldots, i_n \mapsto$ inputs</td>
<td>$\langle t_{\text{root}}; {t_V} \rangle \mapsto o$</td>
<td></td>
</tr>
<tr>
<td>$t_{\text{root}}$</td>
<td>see below</td>
<td>$\langle t_V; {v^i_k} \rangle \mapsto \emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$t_V$</td>
<td>see below</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$t_M$</td>
<td>see below</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

The outputs are produced by iterating over the vertices in a partition, as described by Malewicz et al. [MAB+10]: each vertex may update a private mutable value and send messages to any other vertex. A user-defined partitioning function distributes messages to the appropriate message object, and hence to the appropriate partition in the next superstep. In subsequent supersteps, a $t_V$ task additionally depends on messages from other partitions, which are made available to each vertex via its incoming message iterator.

In the $i$th superstep, the master ($t_M$) task depends on the vote objects ($v^i_k$) produced by each vertex task. The master task aggregates the votes, and uses the result to determine the appropriate action. If all partitions (and hence elements) vote to halt, the master task publishes the job output: a list of the names of the current partitions. Otherwise, at least one vertex has voted to continue the computation, so the master task spawns another superstep, with $n$ vertex tasks, and another master task to which the job output is delegated.

Figure 3.8 shows the state of a dynamic task graph during the second superstep of a Pregel
Figure 3.8: Dynamic task graph for the first two supersteps of a Pregel (Bulk Synchronous Parallel) computation

computation. Note that this model is a slight simplification of Pregel, in that aggregation is not handled. A Pregel computation can additionally compute a global aggregate across all partitions, which is broadcast to the next superstep: this can trivially be achieved by adding an additional output to each vertex task, and \( n \) additional dependencies to the vertex tasks (one per partial aggregate).

Pregel achieves much of its efficiency by storing the partitions in memory between supersteps, in contrast to a chain of MapReduce jobs, which would typically serialise the entire state to a distributed file system between supersteps. Note that the definition of a store (Definition 3) does not mandate a particular storage representation for objects, and different objects may have different storage representations. In the Chapter 4, I will present a system that allows objects to be held in memory for efficient access, which improves the performance of Pregel-style computations.
3.3.3 While programs

Although the previous examples demonstrate the broad applicability of dynamic task graphs, they do not satisfy the effective calculability criterion. Therefore, in order to satisfy this criterion, I will now show a reduction from a Turing-complete language to the dynamic task graph execution model. Furthermore, in order to show that dynamic task graphs are practical, the reduction ensures that, although the amount of computation in each task is bounded, the overall dynamic task graph can represent unbounded computations.

It is well known that most structured programming languages (such as Algol, C, Java and Pascal) are Turing-complete. However, a full translation of a real-world programming language into a dynamic task graph would be laborious and would obscure the salient features that make the model Turing-complete. Fortunately, Böhm and Jacopini proved that a simple “while program” language (based on flowchart notation) is Turing-complete, and requires only the following language features:

**Assignment** The smallest subprogram is the assignment statement, \( x := f(\ldots) \), where \( x \) is the name of a variable and \( f \) is a primitive recursive function that is evaluated with regard to the current execution state (i.e. the values of other variables may be among the arguments of \( f \)). Since \( f \) is a primitive recursive function, its execution time can be bounded by inspection [MR67].

At this point, some additional definitions are necessary. Let \( \text{Var} \) be the set of variable names. Let the set of values be \( \mathbb{N} \). Then the state, \( \sigma \in \text{State} \subset \text{Var} \times \mathbb{N} \), is a mapping from variable names to values. Furthermore, \( \sigma[x \mapsto y] \) is defined to be the same mapping as \( \sigma \) on all variable names except \( x \), which maps to the value \( y \in \mathbb{N} \).

**Sequence** The subprogram \( \Pi(a, b) \) is equivalent to the following sequence:

- Execute subprogram \( a \)
- Execute subprogram \( b \)

**While loop** The subprogram \( \Omega(\alpha, a) \) is equivalent to the following while loop:

\[
\text{while } \alpha \text{ is } 0 \text{ do} \\
\quad \text{Execute subprogram } a
\]

**If statement** The subprogram \( \Delta(\alpha, a, b) \) is equivalent to the following if statement:

\[
\text{if } \alpha \text{ is } 0 \text{ then} \\
\quad \text{Execute } a \\
\text{else} \\
\quad \text{Execute } b
\]

To reduce a while program to a dynamic task graph, each subprogram is transformed into one or more tasks. The first challenge is handling mutable variables. Recall from Definition 3 that each
name in the store of a dynamic task graph can map to at most one immutable concrete object. To work around this limitation, the state before and after executing a task are represented by immutable state objects, \( \sigma_i \). Figure 3.9 illustrates how state objects are passed between tasks in a simple program that assigns to two variables, \( x \) and \( y \). In particular, Figure 3.9(a) shows how the store of the dynamic task graph is updated with new state objects as the assignment statements are executed. Note that the reduction requires fresh names for the intermediate stores (i.e. \( \sigma_1 \) and \( \sigma_2 \) in Figure 3.9); several schemes are possible, and I will discuss a practical decentralised naming scheme in Subsection 4.2.1.

The second challenge is that Böhm and Jacopini’s original conception of a while loop uses a cycle in the corresponding flowchart, and the dynamic task graph must remain acyclic. To address this challenge, it is possible to use the same approach as the master tasks in §3.3.2, which perform a test for termination and spawn another iteration (superstep) in a data-dependent manner. In the case of while loops, a while-loop task evaluates whether \( \alpha = 0 \) (\( \alpha \in \text{Var} \)) in the current store; if so, it spawns a task to execute the subprogram \( a \), and another while-loop task

---

**Figure 3.9:** A simple while program that performs \( x := 6; y := 3; x := 7 \). (a) To enable multiple assignment to mutable variables, the store contains several state objects, \( \sigma_i \). (b) In the dynamic task graph, tasks depend on a state object and may produce a new state object.
CHAPTER 3. DYNAMIC TASK GRAPHS

Figure 3.10: Dynamic task graph for two iterations of a while loop, $\Omega(\alpha, a)$. An $\Omega$ (while-loop) task evaluates the predicate $\alpha = 0$, and, if the predicate is true, spawns a task to perform subprogram $a$ with the current state. Otherwise, it produces its expected output by copying its input state without modification.

Table 3.4: Definition of a dynamic task graph for executing a while program. Note that it is assumed that $\sigma_k$ is a fresh name, generated using a scheme such as is described in §4.2.1. The $\text{TRAV}(a, i)$ function returns the $i^{th}$ element in a post-order traversal of program $a$’s expression tree, which contains $N$ nodes.

<table>
<thead>
<tr>
<th>$t$</th>
<th>$\Sigma_t$</th>
<th>$T_t$</th>
<th>condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Initial job)</td>
<td>$t_{\text{pre}} \mapsto \text{see below}$</td>
<td>$\langle t_{\text{pre}}; {a_0} \rangle \leadsto {\sigma'}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\langle t_{\text{pre}}; {a_0} \rangle \leadsto {\sigma'}$</td>
<td>$a_1 \mapsto \text{TRAV}(a_0, 1)$</td>
<td>$\langle a_N; {\sigma_0} \rangle \leadsto {\sigma'}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\langle a; {\sigma_i} \rangle \leadsto {\sigma_j}$</td>
<td>$\sigma_j \mapsto \Sigma(\sigma_i)[\alpha \mapsto f(\ldots)]$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\langle \Delta(\alpha, a, b); {\sigma_i} \rangle \leadsto {\sigma_j}$</td>
<td>$\sigma_j \mapsto \Sigma(\sigma_i)$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\langle \Omega(\alpha, a); {\sigma_i} \rangle \leadsto {\sigma_j}$</td>
<td>$\sigma_j \mapsto \Sigma(\sigma_i)$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

that depends on the updated state. Otherwise, the task produces its expected output by copying the current store without modification. Figure 3.10 shows the dynamic task graph that results from a while loop with two iterations.

A while program has a finite textual representation, so it may be stored as a program object, $a_0$. The root task of a while program, $t_{\text{pre}}$, preprocesses the program object by performing a
post-order traversal of the program expression tree, which contains \( N \) nodes. The preprocessor recursively creates objects for the child subprograms of the current node, then creates an object for the current node that contains the names of the child objects. The recursion terminates when it reaches an assignment statement. For example, to preprocess the subprogram \( c = \Pi(a, b) \), the preprocessor generates trees of objects for subprograms \( a \) and \( b \), respectively binds the fresh names \( n_a \) and \( n_b \) to roots of those trees, then creates an object containing a textual representation of \( \Pi(n_a, n_b) \) and binds it to the fresh name \( n_c \). A similar approach is taken for \( \Delta \) subprograms (which have two children) and \( \Omega \) subprograms (which have one child). The preprocessor therefore transforms the single program object into a linked tree of subprogram objects.

Each object in the linked tree can serve as the code dependency for zero or more tasks. The \( N^{th} \) object in the post-order traversal corresponds to the root of the tree, and hence the code dependency of the first task (after the preprocessor) to be executed. Table 3.4 shows the complete definition of a dynamic task graph for an arbitrary while program. The computation performed by each kind of task is bounded:

- The preprocessor task, \( t_{\text{pre}} \) performs a post-order traversal of the \( N \) nodes in the expression tree, and produces \( N + 1 \) objects of bounded size, in \( O(N) \) time.

- An assignment task must execute a primitive recursive function, which requires bounded time, and create a copy of the state with the updated binding. Since all variable names must be present in the program text, the total size of the new state is \( O(N) \).

- A \( \Pi \) (sequence) task unconditionally spawns two tasks, which requires \( O(1) \) time.

- A \( \Delta \) (if statement) task must look up the given variable name in the current state and evaluate it (which requires \( O(\log N) \) time), if the state object is sorted by variable name. It then spawns one task (for the appropriate subprogram), which requires \( O(1) \) time.

- An \( \Omega \) (while loop) task must look up the given variable name in the current state and evaluate it (which requires \( O(\log N) \) time, by the argument above). It the either spawns a task for the loop body (which requires \( O(1) \) time) or creates a copy of the current state (which requires \( O(N) \) time).

Therefore, I have shown that dynamic task graphs are sufficiently expressive to represent all Böhm-Jacopini flow diagrams, and hence the model is Turing-complete. Furthermore, because each task performs a bounded amount of computation, the model is practical: it does not rely on a single task (or finite set of tasks) to perform unbounded computation.

The reduction from while programs to dynamic task graphs is useful for pedagogical purposes, but the resulting task graph is rather impractical. In particular, each task performs a negligible amount of work, which means that the execution time would be dominated by task overhead:
in Section 6.2, I will show that—in a distributed implementation of dynamic task graphs—the overhead is approximately 13 milliseconds, which makes this approach very inefficient. Moreover, because of the sequence of state objects threaded between tasks, the execution will always be sequential.

Nevertheless, the reduction presented in this subsection has consequences that will be useful later in this dissertation. First, recall that the assignment statement computes a primitive recursive function. The model can trivially be extended to spawn a (finite) subgraph of tasks—e.g. a MapReduce graph (§3.3.1)—to compute the value for the assignment, which hence achieves parallelism; the simple task programming model (§5.1) facilitates this. Furthermore, the use of state objects to implement a variable store can be extended to include control state, which transforms the state object into a continuation. This is the basis of the distributed thread programming model (§5.3), in which tasks depend on delimited continuations [Fel88, DF90] in order to implement blocking on future events.

### 3.4 Summary

In this chapter, I have defined dynamic task graphs, and demonstrated that they are a highly expressive abstraction for data-flow parallelism. By mapping existing models on to dynamic task graphs, I have shown that dynamic task graphs are capable of computing any effectively calculable function, even when the computation in an individual task is bounded. Furthermore, jobs may incorporate well-known parallel patterns such as MapReduce and BSP, and arbitrary compositions of these patterns.

The expressive power of dynamic task graphs arises from two main features:

**Task spawning** Tasks in a dynamic task graph are permitted to spawn further tasks based on their input data, which means that the number of tasks in a dynamic task graph can be data-dependent. This permits iterative algorithms (in which one task spawns a graph of tasks comprising the subsequent iteration; e.g. BSP, while programs) and recursive, divide-and-conquer algorithms (in which a task spawns multiple similar tasks that operate on subsets of the original task input; e.g. Fibonacci).

**Code as objects** The ability to represent code as data objects permits great flexibility in the behaviour of spawned tasks. For example, it enables higher-order tasks such as those demonstrated in the MapReduce translation. As I will show in Chapter 5, the code-as-objects approach also allows a translation from an imperative programming style to a data-flow program using continuations.

Many execution models have the same expressive power as dynamic task graphs. However, an important advantage of dynamic task graphs arises from the definition of state transitions.
Updating a dynamic task graph is idempotent, which simplifies a distributed implementation when the network is unreliable. Furthermore, the evolution of a dynamic task graph is monotonic, which simplifies coordination [ACHM11], and ensures that all previously-executed tasks may be re-executed. These properties are useful when implementing fault tolerance, as I will discuss in Section 4.4.

Having laid the formal groundwork for dynamic task graphs, the subsequent chapters of this dissertation discuss a concrete implementation of the model, called CIEL. The next chapter discusses how the system achieves efficient performance in a data-intensive scenario.
Chapter 4

A universal execution engine

The core contribution of this thesis is to demonstrate that data-dependent control flow can be supported efficiently in a distributed execution engine. Therefore, in this chapter, I will introduce CIEL, which is a distributed implementation of Chapter 3’s dynamic task graph execution model.

Figure 4.1: The reference lifecycle represents the process of object creation in a CIEL job. The principal aim of a computation is to turn Future references into Concrete references. In addition, Streaming, Sweetheart, Tombstone and Value references serve as non-functional hints that improve the performance and robustness of a computation.

CIEL is a data-centric execution engine: the aim of a CIEL job is to produce one or more objects, using the lazy evaluation algorithm presented in §3.2.2. Since these objects may be very large, CIEL uses references as an indirection that allows objects to be handled without passing the whole object by value. A reference can be thought of as a single name-to-object binding in the dynamic task graph’s store (§3.1). There are several different kinds of reference, as depicted in Figure 4.1; these different types and the transitions between them form the central theme of this chapter.

CIEL is modelled on previous task-parallel distributed execution engines, such as MapReduce [DG04] and Dryad [IBY+07], with additional support for dynamic task graphs; in Section 4.1, I describe the components of a CIEL cluster and explain the distributed coordination between those components. Since CIEL is designed to process large data sets, it includes a
simple distributed storage system, which I introduce in Section 4.2. To achieve high throughput, a distributed execution engine must “put the computation near the data” [Gra08]; therefore CIEL includes a scheduler that is designed to achieve high performance on both simple and iterative data-intensive jobs, and I present and evaluate this scheduler in Section 4.3. Finally, since CIEL is designed to run on commodity computers, it must tolerate faults in any component of the cluster, and I discuss the various fault tolerance techniques that CIEL incorporates in Section 4.4.

4.1 Distributed coordination

CIEL is a system for executing dynamic task graphs: it achieves parallelism by running several tasks in parallel, and dynamism by allowing those tasks to modify the task graph. To motivate the following discussion, this section provides a high-level overview of the CIEL system. First, the components of a CIEL cluster are introduced (§4.1.1), followed by a more-detailed discussion of the task graph representation (§4.1.2). The remaining subsections discuss how a computation is executed at the job level (§4.1.3) and the task level (§4.1.4).

4.1.1 System architecture

Figure 4.2 shows the high-level architecture of a CIEL cluster. Like many previous execution engines [DG04, IBY+07], CIEL has a master-worker architecture, with one or more clients that submit jobs to the system. This subsection outlines the main roles of each component.

Each CIEL cluster has a single master that is responsible for coordinating the execution of a collection of jobs. The master’s state is stored in four tables, respectively storing information about the tasks, objects, jobs and workers in the system. Together, the task table and the object
Table comprise the dynamic task graph of currently-running jobs, and they perform dependency resolution; the receipt of task results updates the contents of the tables, as I will discuss in §4.1.2. The job table contains the currently-active jobs and underlies the main interface to clients: a client may insert a job into the job table, and synchronise with job completion. The worker table contains the details of the workers in the present cluster: when a new worker is added, it inserts a row in the worker table, and that row is deleted when the worker shuts down gracefully or its failure is detected. The scheduler is the principal active component in the master: it identifies runnable tasks and dispatches them to workers, using the techniques described in Section 4.3. The other active component is the recovery manager, which monitors the state of the cluster to identify failed workers, and investigates failed tasks, as discussed in Section 4.4.

The several workers are responsible for executing tasks and storing data in the cluster. The master dispatches tasks to the workers, which execute those tasks and return metadata about the task results to the worker. To support a variety of task implementations, incoming tasks are dispatched to the appropriate executor, which is a generic component that prepares a task for execution, runs the task and marshals the result for sending back to the master. CIEL supports various executors including, for example, executors for Java, Scala and pipe-based programs. Executors provide the programming models for CIEL jobs, which I will discuss in more detail in Chapter 5. The other main component of a worker is the object store, which stores the concrete objects in the dynamic task graph. Since objects are immutable and uniquely named, the object store presents a simple key-value interface. The object store is optimised for local access, since the objects are stored on local disks, but it exposes a remote interface so that other workers can read objects from the store. More details of the storage system are given in Section 4.2.

The role of the client in a CIEL computation is intentionally minimal. The client can upload input data to the workers, and submit jobs to the master. However, once a job has been submitted, the client no longer plays any role in job execution. This greatly simplifies client fault tolerance (§4.4.1).

### 4.1.2 Task graph representation

Recall from Section 3.1 that the dynamic task graph is a bipartite graph of tasks and objects. Therefore, the dynamic task graph can be represented by two tables: a task table and an object table. This subsection discusses the contents of these tables, and how they evolve throughout the execution of a job.

The task table is keyed by task ID, and stores information about every task that has previously been spawned in a particular job. Each value in the task table is a task descriptor, which contains three classes of attribute: essential, informational and transient. The essential attributes

---

1 Although Figure 4.2 represents these as single tables, they are partitioned by job, to facilitate resource allocation policies at the job level (§4.3).  
2 See §4.2.1 for details of how task and object IDs are chosen.
are immutable and are used to define the task: i.e. the code dependency and the set of data dependencies. The informational attributes can be derived deterministically from the essential attributes, but are cached in the task table to aid book-keeping; they include the name of the executor that runs the task, the ID of the parent task (if any) and the names of the task’s expected outputs. Finally, the transient attributes are mutable elements of soft state that correspond to the ongoing execution, including the current run-state and profiling information. In messages between the master and worker, only the essential and informational attributes are included; the worker also sends profiling data back to the master during and after task execution.

The object table is keyed by object ID, and stores information about every object that appears as a task input or output. Since objects may be large, the object table does not store the contents of each object. Instead, the object table stores a reference for each object, which includes the unique object ID and—optionally—metadata that can be used to retrieve the object. There are several kinds of reference, the most common of which are described below:\footnote{In addition, there are four other kinds of reference: sweetheart (used as a scheduler hint, see §4.3.2) stream (used for streaming data between concurrently-running tasks, see §4.2.3), tombstone (used to indicate that an advertised object is no longer available due to worker failure, see §4.4.2) and fixed (used to pin a task to a particular worker, see §5.3.3).}

**Future** Includes no location information. Used to refer to a future object that does not yet exist.

**Concrete** Includes a set of locations for the object, and (optionally) the size of the object in bytes. Used to refer to a concrete object that exists on at least one machine in the cluster.

**Value** Includes the content of the object. Used to refer to small objects, when the size of the object is smaller than the relevant location information.

**Error** Includes an optional reason (error code). Used to refer to the results of tasks that have failed deterministically with a fatal error.

In addition to a reference, the object table stores a task ID. If the reference is concrete, the task ID corresponds to the task that produced the object. If the reference is a future, the task ID corresponds to the task that is expected to produce the object. There may be several tasks in the task table that include the object ID among their expected outputs; in this case, the object table stores the ID of the maximal task that is expected to produce that object (§3.1).

Recall from Section 3.1 that applying the executor function to a task will result in a set of spawned tasks and a store of produced objects. When a task executes, its worker sends spawn and publish messages to the master, which are turned into spawn and publish operations on the task and object tables.

A **publish** operation specifies a task ID and a reference. When a publish operation is applied, it updates the object table with a new reference for a particular object ID. For example, the object table entry for a task result will initially contain a future reference. After the task runs, it will
Table 4.1: Rules for combining an existing reference with an incoming reference. Note that, because Error references arise from deterministic failure, it is not possible for an object to be both concrete and an error, so the invalid cases are denoted by a dash.

<table>
<thead>
<tr>
<th>Existing ref.</th>
<th>Incoming reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Future</td>
<td>Future</td>
</tr>
<tr>
<td>Concrete$_S$</td>
<td>Concrete$_S$</td>
</tr>
<tr>
<td>Value</td>
<td>Value</td>
</tr>
<tr>
<td>Error</td>
<td>Error</td>
</tr>
</tbody>
</table>

publish a concrete reference to the newly-produced object, specifying the location where the object can be found. The publish operation will then “upgrade” the object table entry to contain a concrete reference. Table 4.1 details the rules for combining an existing reference with an incoming published reference. If a publish operation causes a reference to become concrete for the first time, the task ID is also recorded in the task table, which facilitates re-execution in the event that the object is lost due to failure (§4.4.2). Otherwise, if the reference was already concrete, the task ID is discarded. This allows “non-functional” publishes: for example, if a task copies an object to another worker, it may publish a concrete reference that includes the new location, to give the scheduler additional information when scheduling tasks that depend on that object (see §4.3.2 for an example of how this can be useful in iterative computations).

A spawn operation specifies a parent task ID and a new task descriptor. When a spawn operation is applied, it inserts a new entry into the task table, including the code dependency, data dependencies and expected outputs from the task descriptor, the parent task ID, and an initial set of transient attributes. Applying a spawn operation also updates the object table: for each of the task’s expected outputs, a new entry is created containing a future reference and the spawned task ID. The spawned task may have been delegated one or more outputs from its parent (§3.1); in this case, there will already be an object table entry for those outputs (containing a future reference) and its expected producer ID is simply updated to the spawned task ID.

4.1.3 Job execution

A CIEL job involves several steps, each of which entails communication between the components described above. This subsection outlines the steps in a successful CIEL job.

The first step is job submission: the client sends a message to the master containing the details of the new job. Although a job may eventually contain many tasks, it is specified as a single
root task with no dependencies and a single output, which when run spawns the tasks that make up the job. This allows the description of a job to be very simple: Figure 4.3 shows a job descriptor that contains the specification for a job that estimates the value of π. (This algorithm is a simple MapReduce-style computation, which is evaluated and described further in §6.3.1.) The job descriptor contains three stanzas: the first is the package, which collects together input files and provides a location-independent symbolic name for each input (script and jar in Figure 4.3) that may be used within the job. The second stanza defines the root task, including the name of its executor, the initial task arguments, and any environment that should be passed into the task. The example in Figure 4.3 shows an initial task that uses the Skywriting executor (§5.3.1), invokes the packaged file named script, and sets the N_TASKS and N_SAMP environment variables with values from the invoker’s environment, or default values if these are not available. The final stanza is optional, and contains job-level options; in Figure 4.3 the random scheduler is selected, because this particular job does not require worker-affinity. The client uploads all of the files in the package to the cluster, which translates them into references (§4.2), submits the root task to the master as a new job, and optionally blocks until the job has completed.

The root task has one expected output, which corresponds to the result of the whole job. The role of the master is to evaluate that job output, which it does by executing tasks. Initially, when the master receives a new job, there is a single entry in the task table, corresponding to the root task. Since—by definition (§3.1)—the root task has no unfulfilled dependencies, it immediately becomes runnable. The scheduler selects an idle worker6 on which to run the task, and dispatches the task to that worker. When it executes, the root task will spawn a list of tasks and publish a list of objects, and send these back to the master.

The operation of the master and worker thereafter is summarised by the pseudocode in Figure 4.4. The master scheduler dispatches runnable tasks to idle workers, using a job-specific

---

6In general, the root task does not perform data-intensive processing, so it can be dispatched to any worker.
j ← receive new job from client

\[ t_{\text{root}} \leftarrow \text{root task of } j \]

\[ o \leftarrow \text{expected output of } t_{\text{root}} \]

\[ \text{insert } t_{\text{root}} \text{ into the task table} \]

\[ \text{while } o \text{ has not been produced do} \]

\[ \text{while there is an an idle worker, } w_{\text{idle}}, \text{ and a runnable task, } t_{\text{runnable}} \text{ do} \]

\[ \text{dispatch } t_{\text{runnable}} \text{ to } w_{\text{idle}} \]

\[ \langle \Sigma_{\text{published}}, T_{\text{spawned}} \rangle \leftarrow \text{receive next task result from worker } w_{\text{done}} \]

\[ \text{update the object table with } \Sigma_{\text{published}} \]

\[ \text{update the task table with } T_{\text{spawned}} \]

(a) Master

\[ \text{while forever do} \]

\[ t \leftarrow \text{receive next task from master} \]

\[ \text{execute task } t \]

\[ \Sigma_{\text{published}} \leftarrow \text{references published by } t \]

\[ T_{\text{spawned}} \leftarrow \text{tasks spawned by } t \]

\[ \text{send } \langle \Sigma_{\text{published}}, T_{\text{spawned}} \rangle \text{ to the master} \]

(b) Worker

Figure 4.4: Pseudocode for the CIEL (a) master and (b) worker components.

scheduling policy to match tasks and workers (§4.3). The result for a particular task is a list of spawned tasks and a list of published objects, either—but not both—of which may be empty. When a worker notifies the master that it has completed a task, it sends those lists, which the master applies to the task graph. As the task graph is updated, either more tasks will become runnable, or the job result will be published, which terminates the job. When the job terminates, the master notifies the client, if it is waiting for job completion.

4.1.4 Task execution

To execute a task, the master dispatches the task to an idle worker. The worker contains one or more executors, which are a concrete implementation of the abstract execution function, \( \mathcal{E} \), defined in Section 3.1: i.e. they transform the set of objects on which the task depends into published objects and newly-spawned tasks.

The executor abstraction enables CIEL to support a wider variety of tasks than existing execution engines, while maintaining a simple coordination layer. CIEL achieves this goal by storing

\[ ^7 \text{Recall from } \S3.2.1 \text{ that a task has at least one expected output, and it must fulfil all of its expected outputs by either producing that output or delegating it to a spawned task. Therefore, the result must contain at least one published object or spawned task.} \]
task arguments in an opaque object, and representing them as the task’s code dependency. From
the master’s point of view, the code dependency is merely one object—a finite-length sequence
of bytes—among the set of objects on which the task depends, and it may or may not already
exist. The code dependency is only interpreted when the task is dispatched to the appropri-
ate executor on a worker and the code object is retrieved. This late binding allows complete
flexibility in the implementation of the code dependency.

As a concrete example, consider a simple task that is implemented in the Java programming lan-
guage. To define this task, one must specify the name of an entry-point class, and a “classpath”
comprising one or more Java archive (JAR) files that contain the bytecode implementations of
the necessary classes [Ora]. Additionally, let us assume that the task takes two other objects
as data inputs, and the classpath contains two JAR files. As a CIEL task, this would be rep-
resented as a task with a set of five dependencies, as shown in Figure 4.5. In this case, the
code dependency contains a key-value dictionary (stored in, for a concrete example, JavaScript
Object Notation (JSON) [Cro06]) in which the jar_lib key maps to the list \([m, n]\), and the
inputs key maps to the list \([x, y]\), and the class key names the task implementation class.
This dictionary is only interpreted by the executor, which instantiates a Java virtual machine,
dynamically loads the classes in the JAR files and dynamically instantiates the named class.

The second advantage of the executor abstraction is that the code dependency need not have
a simple key-value representation, because only the executor need be able to interpret it. For
example the object may be a serialised representation of an object in an arbitrary programming
language’s object model, such as a Java object, which enables the programmer to use the type
system of that language to specify a task. Furthermore the code dependency may itself be
executable: the distributed thread programming model (described in the following chapter) uses
continuation objects to represent the state of a blocked thread, which allows a single logical
thread of execution to extend across multiple tasks.

**Figure 4.5:** Representation of a Java example task, showing the contents of the code de-
pendency object in JSON notation.
I will defer discussion of how tasks are programmed until Chapter 5, and focus here on the interaction between an executing task and the master. As described in Subsection 3.2.1, tasks are pure functions, but they can have two observable side effects on the dynamic task graph: object publication and task spawning. These side effects are idempotent, which ensures that it is safe to re-execute a task in the event of a fault (§4.4.2).

If a task wishes to pass data to another task, it must be encapsulated in a CIEL object. A task creates an object by choosing a fresh name (according to the scheme defined in §4.2.1) and writing it into the local object store. After creating the object, the task must publish a reference to that object by sending a message to the master, which adds the object to the dynamic task graph, and thereby notifies any other tasks that depend on it. The publish message includes the ID of the publishing task, which associates the object with the task in order that it can be regenerated in the event of a failure (§4.4).

Similarly, a task can spawn a child task by sending a spawn message to the master, which contains a task descriptor as described above. Note that spawning a task is almost always preceded by publishing the object that will be the task’s code dependency.

Since task execution is atomic and non-blocking, and the publish and spawn messages are asynchronous, the executor can buffer all messages until after the task terminates, which eliminates the overhead of sending many small messages from a task that spawns a large number of tasks. The buffering implementation must ensure that an object is published before spawning any task that depends on that object; otherwise the spawn would be rejected because the task could never become runnable. The current master implementation applies all publish operations to the dynamic task graph before applying all spawn operations.

### 4.2 A simple distributed store

A guiding principle of commodity data-intensive systems is that they use distributed storage because a computation can use hundreds or thousands of disks in parallel, and thereby achieve much higher throughput than would be achievable with a centralised store. For example, Google MapReduce uses the Google File System (GFS) [DG04, GGL03], Hadoop MapReduce uses the Hadoop Distributed File System (HDFS) [SKRC10], and Dryad uses the Cosmos Storage System [CJL+08] or TidyFS [FHIS11]. Each of these systems is optimised for storing large sequential files that are either immutable or provide append-only access. In CIEL, the user-level atomic storage unit is the object. This section introduces the storage architecture, which is designed to achieve high parallel throughput when accessing objects.

The storage subsystem in CIEL mirrors several previous distributed storage systems, such as GFS [GGL03], HDFS [SKRC10] and Ursa Minor [AEMCC+05], which store potentially-
CHAPTER 4. A UNIVERSAL EXECUTION ENGINE

replicated chunks of data identified by a flat name\(^9\). All CIEL objects are *immutable* and have a unique name, which means that two objects in different locations with the same name are defined to have identical contents. Subsection 4.2.1 discusses how object names are chosen. The objects are distributed across cluster nodes (i.e. the workers) and location metadata is stored in the dynamic task graph in the master. Subsection 4.2.2 discusses how objects are stored and accessed. Finally, CIEL also supports a limited form of mutation, whereby tasks may produce and consume streams of data; Subsection 4.2.3 discusses how this feature is implemented.

### 4.2.1 Naming objects

Recall that all objects in an CIEL computation have a *unique* name (§3.1). Furthermore, objects may be created dynamically as a job executes, so the set of names used by a task cannot trivially be pre-determined. Therefore, there must be a mechanism for a task to choose a new name for an object. In addition, that mechanism must be *deterministic*, because—in order to provide transparent fault tolerance, as discussed in Section 4.4—it must be possible to re-execute a task and create exactly the same objects with the same names. This subsection discusses how object names are chosen in a CIEL job.

Object names are chosen at two points in a CIEL job:

1. When a new task is spawned, in order to name the task’s expected outputs; the task’s ID is chosen at the same time.
2. When a task creates a new object, in order to pass data to a subsequent task.

A simple solution would be to name each object using the name of the *task* that produced it (or was first expected to produce it, since outputs may be delegated). In MapReduce and Dryad, this is straightforward, because the task graph is static and so each task can be assigned a unique name when the job is submitted. The situation is more complicated in a dynamic task graph, since the complete set of tasks is not known in advance. Furthermore, maintaining a centralised counter is non-trivial, because parallel tasks may increment the counter in a non-deterministic order, and furthermore a shared counter would represent a bottleneck. Therefore, a decentralised solution is preferable, and a straw-man solution would be to name each task based on the task that spawned it:

<table>
<thead>
<tr>
<th>Parent task ID</th>
<th>Delimiter</th>
<th>Spawn counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. 0:23</td>
<td>:</td>
<td>7</td>
</tr>
</tbody>
</table>

\(^9\)Although the object namespace is flat in these systems, a hierarchical namespace can be built using a separate metadata server, which may store the metadata themselves as objects [AEMCC\(^+\)05].
The example shows the ID of the 7th task spawned by the 23rd task spawned by the root task (arbitrarily defined to be task 0; this could alternatively be a unique job ID).

The obvious limitation of this scheme is that the length of a new task name will grow as an iterative job executes. To achieve a similar result with fixed-length names, one can apply a fixed-length hash function, $H$, after computing the name by the above scheme:

<table>
<thead>
<tr>
<th>Parent task ID</th>
<th>Spawn counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. c9343eb2b95687f324fa39f2fd93d7ea54e7db97</td>
<td>7</td>
</tr>
<tr>
<td>$= H(c9343eb2b95687f324fa39f2fd93d7ea54e7db97:7)$</td>
<td></td>
</tr>
<tr>
<td>$= 1a52cf9ac049b2f4bec33de6ad1bd6f6b15991c1$</td>
<td></td>
</tr>
</tbody>
</table>

The current implementation of CIEL uses the 160-bit SHA-1 hash function [EJ01] to compute fixed-length task names. Although it is theoretically possible that hashing the names of two different tasks may result in the same name, this is highly improbable in practice [QD02].

Given the above scheme for computing task names, the names of a task’s expected outputs can be computed by appending a delimiter and an integer index to the ID of a spawned task:

<table>
<thead>
<tr>
<th>Task ID</th>
<th>Delimiter</th>
<th>Output index</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. a92048c772cc095eb0d20cedc4e3e2e43dd4ab28</td>
<td>:</td>
<td>4</td>
</tr>
</tbody>
</table>

If the task produces additional objects, these can be named using the same scheme, but using a serial number that starts with an integer that is greater than the largest output index.

However, a better object naming scheme is made possible, by observing that the outputs of a task are a deterministic function of the task dependencies. Combined with the use of lazy evaluation, this creates the opportunity to exploit memoisation if two tasks are spawned with the same arguments. When a task is spawned, the task arguments are passed to the relevant executor, which generates a cryptographic hash of the arguments that have an effect on the final output. The actual object naming scheme that CIEL uses is described below.

To obtain a name that is useful for memoisation, the executor attempts to identify the precise dependencies [HLY00] that affect the task result. For example, one useful simplification is that any references in the arguments may be reduced to their name, because the present type (e.g. future, concrete, etc.) of the reference must have no effect on the task output; a task that behaves otherwise would be non-deterministic. In addition, the executor can disregard non-functional arguments—including, for example, directives to stream task outputs (§4.2.3) or cache the inputs (§4.3.2)—which do not affect the task outputs. Having identified the precise dependencies, the executor then uses the SHA-1 hash function to generate a unique 160-bit identifier for the task outputs, and appends the respective output indices to the resulting identifier.

Opaque identifiers, more than 40 characters long, are obviously less user-friendly than the hierarchical filenames used in distributed file systems. To address this problem, the package used
for job submission (§4.1.3) allows the client to associate a job-private symbolic name with an
object that is loaded into the cluster. Furthermore, an object may have recursive structure: to
deal with large distributed files, CIEL uses a symbolically-named index object, which contains
references to each chunk of the overall file.

4.2.2 Storing objects in the cluster

An object becomes concrete when at least one copy of it is stored in the cluster. Recall from
Section 4.1 that each worker has an object store. This subsection briefly discusses how objects
are stored in a CIEL cluster, and how data are loaded into the cluster.

First, it is necessary to elaborate upon the structure of a concrete reference. A concrete reference
contains the following metadata about an object:

**Name** The unique name of the object, which is assigned using the memoisation-based scheme
outlined in the previous subsection. (Fresh uploads are assigned a universally-unique ID
by the master.)

**Location hints** A set of one or more network locations where the object is stored. These
network locations must correspond to workers (or other components that implement the
worker remote read protocol, described below).

**Size hint** (Optional.) The size of the object in bytes.

The location hints provide the means of retrieving an object. In the simplest case, all of the
workers at the network locations specified in the location hints must respond to a request for
the named object with the contents of that object. In the current implementation, this request
is made using an HTTP GET request to a URI constructed from the network location (host and
port) and the object name. The size hint provides a simple consistency check for the receiver,
and it would be trivial to add a checksum to the concrete reference if the storage or transmission
medium were unreliable.

As with chunks in GFS [GGL03], each object in an object store is stored as a separate file in a
UNIX directory, with the same name as the object. The file system provides desirable semantics
for use in the block store. The `stat()` system call can be used to test whether or not an object
exists. When an object is created, it is first stored in a temporary file, then the `link()` system
call is used to write the complete object file into the object store atomically. The object store
provides this functionality as a library.

Concrete objects can be created in at least three ways:

1. When a task produces its output. Unlike MapReduce [DG04], the default behaviour of a
   CIEL task is to write a single copy of the task output to the local object store, because it
Figure 4.6: Expected load distribution using two different replica placement strategies. In this experiment, 100 objects are assigned to 100 workers, with triple-replication. Using two random choices achieves a better load distribution than a single random choice.

is assumed that failures are rare and fault tolerance mechanisms can be used to recover lost objects (§4.4). However, the remote upload facility (described below) may be used to replicate an object across multiple workers.

2. When a task fetches an input from another worker. In this case, the object store acts as a read-through cache. The running task updates the concrete reference for that object, and may publish that reference so that subsequent tasks may be scheduled on that worker. This is useful for iterative algorithms, and its effectiveness is evaluated in §4.3.2.

3. When a client uploads data into the cluster for processing. The current version of CIEL provides two utilities for loading objects into a cluster: a stand-alone data loader and the job submission script. Both utilities support push-based uploading (whereby the client posts objects in one or more chunks to the workers) and pull-based uploading (whereby the client posts URIs to the workers, and the workers fetch from these URIs in parallel to achieve greater I/O throughput).

The object loading utilities obtain a list of current workers from the master, and all object placement decisions are made locally at the client\textsuperscript{10}. The most common policy for selecting replica locations is to select workers at random [TDN11], perhaps subject to constraints such as “at least one replica of a block should be stored on a different rack” [SKRC10]. However, for a single job, this may lead to poor performance, because selecting locations uniformly at random is expected to lead to poor load balance ($\log n / \log \log n$ for $n$ objects assigned to $n$ machines) [ACMR95]. Figure 4.6(a) shows the simulated load distribution that results when as-

\textsuperscript{10}This implies a cooperative relationship between the users of a cluster, since a misbehaving client could perform a denial of service attack against a target worker by uploading a large amount of data to that worker. A simple countermeasure would be for the master to grant the client a set of capabilities that allow it to upload up to a maximum number of bytes to each worker.
signing 100 triply-replicated objects to 100 workers: 5% of workers are assigned no objects at all, which means that any tasks run on those workers will certainly have to fetch their inputs from remote workers, and, depending on the scheduler, it is likely that more tasks will ultimately run non-local. A well-known better strategy is to make two random choices and select the worker to which fewer blocks have been assigned [ACMR95]. Figure 4.6(b) shows the simulated load distribution that results when using this strategy in the same scenario as before. When using two random choices, 47% of workers are assigned the expected number of objects (i.e. three), and only 0.4% of workers are assigned no objects at all. The result of using two random choices is that CIEL is able to schedule more data-local tasks for MapReduce-style jobs on large partitioned files. Section 4.3 discusses the performance consequences of non-local tasks.

### 4.2.3 Streaming objects between tasks

The earlier explanation of task execution (§4.1.4) stated that a task produces data objects as part of its result. This definition implies that object production is atomic: an object either exists completely or not at all. However, since data objects may be very large, there is often the opportunity to stream the partially-written object between tasks, which can lead to pipelined parallelism.

In CIEL, streaming is designed to be a non-functional performance optimisation, which means that the semantics of a job are unaffected by the transport mechanism used at runtime. In all versions of streaming, if a producing task has streamable outputs, it sends a publish message to the master, containing stream references for each streamable output. Table 4.2 shows the object table update rules that are applied to an incoming stream reference: in general a stream reference is more preferable than a future reference, but it is ignored if there is any other type of reference that represents a full copy of an object. Furthermore, an existing stream reference is preferred over an incoming stream reference, on the assumption that a stream that has existed for longer is likely to have produced more data. Stream references may unblock other tasks:

<table>
<thead>
<tr>
<th>Existing ref.</th>
<th>Incoming reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Future</td>
<td>Stream&lt;sub&gt;y&lt;/sub&gt;</td>
</tr>
<tr>
<td>Concrete&lt;sub&gt;S&lt;/sub&gt;</td>
<td>Concrete&lt;sub&gt;S&lt;/sub&gt;</td>
</tr>
<tr>
<td>Value</td>
<td>Value</td>
</tr>
<tr>
<td>Error</td>
<td>—</td>
</tr>
<tr>
<td>Stream&lt;sub&gt;x&lt;/sub&gt;</td>
<td>Stream&lt;sub&gt;x&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

Table 4.2: Rules for combining an existing reference with an incoming stream reference. A stream reference, Stream<sub>y</sub>, is currently being produced at location <i>y</i>.

---

11The streaming implementation contains optimisations developed by Christopher Smowton, including producer-to-consumer notifications and direct TCP streaming.
the stream consumers. A stream consumer executes as a normal task, but the executed code must include logic that blocks to wait for the stream producer if it reaches the end of the current input.

As I will discuss in Chapter 5, many executors expose a stream-based interface for reading (and writing) objects. Therefore, in languages such as Java and Python that support polymorphic stream or “file-like” objects, support for streaming execution is implemented transparently in the language bindings. Where it is not possible to modify the stream implementation—for example, in C programs that use file descriptors and expect blocking file semantics, or legacy executables that require an input filename—CIEL creates a named pipe that is passed to the task for reading, and a worker thread writes incoming data into the pipe.

HTTP is the default transport protocol for fetching both streaming and concrete references. A concrete reference is fetched by making a single HTTP GET request to one of the locations in the reference’s location hints, which may be chosen based on proximity in the network or else at random. By contrast, a streaming reference is fetched by making multiple HTTP GET requests to the stream producer and (ab)using the HTTP Content-Range header to indicate that only part of the object is available. The fetched data is written to a temporary file in the object store, and into a task-input pipe if necessary. To avoid distributed polling, the consumer subscribes to notifications from the producer, which are sent when the object grows by a configurable number of bytes. When the producer signals that the object has been completely written, the consumer atomically inserts the consumed object into the consumer’s object store, and, if necessary, closes the task-input pipe.

The advantage of HTTP streaming is that it does not couple the execution producer and consumer tasks. The producer is not coupled to the consumer: it can continue to write data to the local disk without blocking. Furthermore, multiple consumers can read the same stream, and they may join at any time. However, the performance is not optimal because, if the consumer is slower than the producer, the data will be written to and read from disk twice—once at the producer, then again at the consumer.

CIEL also supports direct TCP streaming in certain circumstances. In this case, CIEL connects the producer and consumer tasks using a TCP socket connection: the producer sends data as it is produced into the socket, and the consumer receives data directly from the socket. To enable this mode, the programmer must provide the following hints when creating both the producer and the consumer:

- The producer must be created with the single_consumer flag, which indicates that only one task will attempt to read the task output.
- The consumer must be created with the no_store flag, which indicates that the fetched data should not be written to disk at the consumer.

Direct TCP streaming requires that the producer and consumer tasks be assigned to workers at the same time; however, there may not be sufficient cluster capacity to ensure this. To avoid
deadlock, the producer task waits for a short period of time (e.g. 5 seconds) before writing any data. If the consumer does not connect to the producer within that period, the producer redirects its output to disk, and the consumer will fall back to using an indirect streaming mode.

The streaming facility in CIEL is designed as an optional optimisation. In Section 6.6, I will show how it can be used to parallelise the binomial options pricing model, which contains fine-grained data dependencies and pipelined parallelism. However, CIEL is designed to adapt to varying cluster membership, and so the correct functioning of an CIEL computation should not depend on the number of available workers. Therefore, no stream producer is ever blocked indefinitely to wait for a consumer, and every stream consumer can consume from a producer that has run to completion. One consequence of this is that CIEL does not support infinite data streams, and I will discuss this limitation in Chapter 7.

### 4.3 Scheduling

Once a task becomes runnable, the scheduler in the master either chooses a worker on which to run the task, or queues the task for later execution. CIEL supports a variety of scheduling policies, and this section focuses on scheduling policies for data-intensive jobs.

Given a fixed set of resources, the general scheduling problem for an acyclic data-flow graph is the minimum makespan problem, which is known to be NP-hard [Ull75]. However, a practical scheduler implementation must also address the following practical challenges:

- The execution time of a task may not be known in advance.
- The execution time of a task may depend on the worker on which it runs. For example, the workers may have heterogeneous processing resources.
- The execution time of a task may depend on the other tasks that are executing at the same time. For example, if a task fetches data from another worker, the other tasks running on that worker may be adversely affected.
- In CIEL, the full dynamic task graph is data-dependent. This makes it impossible to compute an optimal schedule in advance, because, for example, the total number of tasks is not known.

As a result, computing an optimal schedule is both computationally and practically intractable, and CIEL follows existing distributed execution engines in using a variety of heuristics to achieve acceptable performance [ZKI+08, IPC+09, ZBSS+10, AKG+10, AAK+11]. The remainder of this section describes two such heuristics used in CIEL, which improve the performance of data-intensive algorithms in general (§4.3.1) and iterative algorithms in particular (§4.3.2). In addition, the effects of different scheduling policies are evaluated in detail for an iterative computation (§4.3.3).
4.3.1 Locality-aware scheduling

The simplest scheduling policy would involve adding each task, as it becomes runnable, to a global queue from which workers remove tasks as they become idle. This is the classic task farming model, as described in Section 2.3. While its simplicity is attractive, the single queue model is not appropriate for a data-intensive system, because it does not take into account data locality. In a data-intensive computation, most tasks have at least one large input [IBY+07, DG04], and therefore Gray’s maxim applies: “Put the computation near the data” [Gra08]. To achieve this objective, the CIEL scheduler is locality aware, which means that it attempts to schedule each task on a worker that contains the largest proportion of its input data.

The CIEL scheduler is non-preemptive and uses multiple queues, similar to the scheduler implementation in Hadoop [Had]. The master holds one queue for each worker, and a single global queue\(^{12}\). Tasks may be added to zero or more per-worker queues, and all tasks are added to the global queue, in order to permit work stealing when a worker exhausts its local queue.

When a task becomes runnable, the scheduler analyses the task’s dependencies to choose the workers in whose queues the task will be enqueued. Since each dependency can be resolved to a concrete reference\(^{13}\)—which includes a set of location hints and a size hint (in bytes)—the analysis simply computes the total number of input bytes stored on each worker. A naïve approach would simply choose the worker with the largest number of bytes locally. However, as mentioned in Section 4.2, many job inputs are replicated, which means that several workers may be equally good candidates for execution. Furthermore, the exact amount of data stored on each worker may differ slightly: for example, consider a task that depends on a one-gigabyte object that is replicated on workers \(a, b\) and \(c\), and a one-megabyte object that is stored on worker \(a\). Clearly, although worker \(a\) is the best candidate, workers \(b\) and \(c\) are approximately “as good as” worker \(a\), and certainly preferable to worker \(d\), which stores neither object. As a result, CIEL uses an effective locality threshold, \(\theta\), to choose candidate workers: any worker that stores at least \(\theta\%\) of the input data relative to the best candidate is deemed also to be a candidate. The current prototype is optimised for MapReduce-style jobs where a map task’s input is approximately 64 MB, and the job implementation (libraries etc.) may be up to 6 MB. Therefore, setting \(\theta = 90\%\) gives the expected behaviour, whereby all replicas are equally preferable even in the case that one worker holds both the data and the code, because \(\frac{64+6}{70} > 90\%\).

Once all runnable tasks have been assigned to queues, the scheduler assigns tasks from the queues to each of the idle workers. The scheduler first attempts to remove a runnable task from a worker’s local queue and assign that task to the worker. If this succeeds, the scheduler marks the task as assigned (in the task table), which leads to the task being discarded from other

---

\(^{12}\)The addition of more levels of queues, for example per-rack queues [IPC+09], is a trivial extension of this scheme.

\(^{13}\)If a dependency can only be resolved to a stream reference, that reference has no impact on the scheduling decision.
Fig. 4.7: CIEL achieves over 80% data-locality for a fixed problem size on up to 100 workers. This graph shows the proportion of data-local tasks when executing the grep benchmark, which comprises 166 parallel tasks and is described further in §6.3.2.

This simple approach allows CIEL to achieve a high proportion of data-local tasks when executing MapReduce-style jobs. Figure 4.7 shows the proportion of tasks that run on local data in the grep benchmark executions, the configuration of which is discussed in more detail in Subsection 6.3.2. In this experiment, the input is a 22 GB text file, split into 166 objects containing approximately 128 MB each. The objects are triple replicated, and loaded into the cluster using the two-random-choice policy described in Subsection 4.2.2. For all cluster sizes, the average proportion of data-local tasks in each job is over 80%. Note that the proportion of data-local tasks is higher for smaller clusters: all tasks are trivially data-local in a single-worker cluster, but—as more workers are added—the load imbalance grows, and it becomes more likely that some workers will have less data-local work to do.

### 4.3.2 Scheduling iterative jobs

In an iterative job, the CIEL scheduler can re-use information about previous iterations to improve the schedule. In many data-parallel iterative algorithms, such as $k$-means clustering and PageRank, the majority of the input data is loop invariant. In CIEL, the first iteration is scheduled using the locality-aware strategy described in the previous subsection. The key observation is that the schedule for one iteration is a good starting point for the next iteration. This subsection describes how this observation is used to improve the performance of iterative jobs.
As discussed in §4.2.2, the object store acts as a read-through cache. Therefore, as an iterative job executes, the execution of non-data-local tasks will cause input data to become available on more workers, which increases the opportunity for data-local tasks in subsequent iterations. To signal these opportunities to the scheduler, when it fetches an object for the first time, the worker sends a publish message containing the new network location, and it is added to the relevant entry in the object table, as discussed in Subsection 4.1.2. Figure 4.8 shows how the distribution of replication factors of the inputs to an iterative job evolves from the first to the hundredth iteration. As the job executes, objects become more replicated as they are copied to new workers for non-data-local tasks. Although one would expect this to lead to better schedules due to more opportunities for data-local tasks, in the limit this strategy leads to every object being replicated on every worker, which is highly wasteful of cluster storage.

Merely increasing the opportunities for data-local tasks is insufficient, because the greedy algorithm for selecting a local task is indifferent between “equally data-local” tasks, and does not select tasks based on maximising the opportunities for other workers to execute data-local tasks. However, after the first iteration, such a matching must exist, because every task must either have been data-local or fetched its inputs to a new worker, on which the same task would subsequently be data-local. On an uncontended cluster, re-using the same matching would yield performance at least as good as the previous iteration, since all tasks would be data-local and no task inputs would need to be fetched remotely. Therefore, each task would execute in at most the same amount of time as in the previous iteration (modulo performance variance in accessing the disk). Note that the matching may still not be optimal, since there may still be opportunities for a worker—now able to complete its assigned work in a shorter time—to steal work from other workers. Furthermore, simply memoising the previous schedule would lead to inefficient
CHAPTER 4. A UNIVERSAL EXECUTION ENGINE

<table>
<thead>
<tr>
<th>Existing ref.</th>
<th>Incoming reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Future</td>
<td>Sweetheart(y,T)</td>
</tr>
<tr>
<td>Concrete(S)</td>
<td>Sweetheart(y,T)</td>
</tr>
<tr>
<td>Value</td>
<td>Value</td>
</tr>
<tr>
<td>Error</td>
<td>—</td>
</tr>
<tr>
<td>Stream(x)</td>
<td>Sweetheart(y,T)</td>
</tr>
<tr>
<td>Sweetheart(x,S)</td>
<td>Sweetheart(y,S\cup T)</td>
</tr>
</tbody>
</table>

Table 4.3: Rules for combining an existing reference with an incoming sweetheart reference. A sweetheart reference, \(\text{Sweetheart}_{y,T}\) has preferred location \(y\) and set of alternative locations \(T\).

To solve this problem, the CIEL scheduler uses *sweetheart references*, which are a subtype of concrete references that additionally store a single *preferred location* for each object. As discussed in §4.2.2, when a task fetches an object from a remote worker, it may send an informative publish message that indicates the additional location for the object. The programmer may opt instead to send a sweetheart reference, containing the local worker as a preferred location, which updates the object table using the rules in Table 4.3. Note that there is only one preferred location for each object—i.e. the most recently received location—which reduces choice for the scheduler and consequently simplifies the scheduling problem.

The CIEL scheduler takes sweetheart references into account when assigning tasks to worker queues, using a simple heuristic. When the number of input bytes on each worker is totalled, the preferred locations for a sweetheart reference is attributed the number of bytes multiplied by a boost factor (100 in the current implementation), which ensures that a task depending on a single large object with a preferred location will be added to the queue for the corresponding worker. If a task depends on two or more sweetheart references with different preferred locations, each location receives a boost in proportion to the amount of data stored at that location, and the effective locality threshold is used to determine the best worker or workers. Furthermore, if the preferred location for an object becomes unavailable due to worker failure, the scheduler can fall back to alternative locations, because the sweetheart reference retains information about all known replicas of an object.

The sweetheart reference can also be used to exploit in-memory caching. Several systems have demonstrated the benefits of in-memory caching for iterative algorithms that have a large amount of loop-invariant input data [ZCF+10, PL10, ELZ+10]. Many CIEL executors support a *soft cache*, which allows a deserialised representation of an object (optionally subject to additional processing) to be stored in memory for later re-use by other tasks. Though the implementation of the soft cache is executor-specific, it relies on the worker’s generic ability to maintain an executor process (such as a Java virtual machine) between task invocations.
To use the soft cache, after consuming an object for the first time, a task *puts* the object into the soft cache. The cache entry key is derived from one or more object IDs and an application-specific tag, which allows the cache to store structures that have been derived from more than one object, and multiple different structures that have been derived from a single object. A subsequent task may try to *get* the same key from the cache: note that this operation is non-deterministic, because a subsequent retry of the operation may have a different result, depending on both the worker on which it runs and the cache replacement policy. Therefore, the application programmer must be careful to ensure that the semantics of the task are unaffected by a cache hit or miss. To aid the programmer, the executors provide transparent abstractions, such as the Java *CachingInputStream*, which reads a raw object either from disk or from an in-memory buffer, and presents the same interface to the programmer. At a higher level, the *CachingIterable<T>* provides a type-safe interface for MapReduce-style computations over sequences of records, and further removes the need to deserialise the cached objects.

At the scheduler level, the soft cache works in conjunction with sweetheart references. For large objects, the task that puts the object in the cache also publishes a sweetheart reference to the local worker. Note that the soft cache and sweetheart references are independent, so small broadcast objects (for example, the cluster centroids in a $k$-means computation) can be cached at multiple workers without the scheduler necessarily preferring a particular worker. The cache replacement policy is executor specific, and the current implementation (for Java) uses a *not-recently-used* policy, whereby cache entries are downgraded to soft references after they have not been used for a fixed time period. A softly-referenced object is vulnerable to garbage collection if memory pressure grows beyond a certain level [CLK99, pp. 816–820], but there remains a possibility that a softly-referenced object will not be collected and result in a cache hit. Executor processes are also subject to garbage collection within the worker: if an executor process has not been used after a fixed period, it is terminated. Long-lived executor processes are discussed further in §5.3.3.

### 4.3.3 Comparing scheduling policies

To demonstrate the efficacy of sweetheart references and the soft cache, I conducted a simple experiment using $k$-means clustering as an example of a typical iterative algorithm. The details of the algorithm and the experimental configuration are given in Chapter 6; however, there are two pertinent details that are necessary to understand the current experiment:

- The majority of the input data to each iteration is invariant, and comprises several large objects, which may be processed in parallel. In this experiment, the input comprises 100 large objects, each containing 128 MB of dense floating-point vectors, which are triply-replicated across 100 workers. In the versions using sweetheart references, each task publishes a sweetheart reference for its respective input.
Figure 4.9: Number of non-local tasks in the first 20 iterations of $k$-means using the default scheduler or sweetheart references. In total, each iteration comprises 100 independent tasks.

- The amount of computation per byte of input is directly proportional to the parameter $k$, which represents the number of clusters to be gathered. In this experiment, one configuration uses $k = 1$ to represent an I/O-bound workload, and the other uses $k = 100$ to represent a moderately CPU-bound workload.

Figure 4.9 shows how the number of non-local tasks evolves over the first twenty iterations of the $k$-means algorithm. In the first iteration, both the default policy and the sweetheart-reference versions have the same distribution of non-local tasks ($\mu = 13.6, \sigma = 2.2$), because they both use the same two-random-choice placement policy described in §4.2.2. Without any caching—as is the case in Hadoop—this proportion would remain constant throughout the entire computation. However, under CIEL’s default policy (Figure 4.9(a)), the number of non-local tasks decreases in subsequent iterations, because more replicas of each object become available (i.e. on the workers that execute non-local tasks); in the 20th iteration, the average number of non-local tasks drops to 5.6 ($\sigma = 1.7$). This “improvement” occurs because, over the first 20 iterations, an average of 171 non-local tasks execute, causing 171 additional object replicas to be stored in the cluster—an increase in storage consumption of $\frac{171}{300} = 57\%$. By contrast, using sweetheart references (Figure 4.9(b)) leads to all tasks being local from the second iteration onwards, with increased storage consumption of only $\frac{13.6}{300} = 4.5\%$.

Figure 4.10 shows the effect of that increase in data-local tasks on execution time. In general, the increase in data-local tasks does not noticeably improve the performance when using the default scheduling policy, because a $k$-means computation must wait for all tasks to complete, which means that even a single non-local task will delay the entire iteration. By contrast, using sweetheart references leads to a substantial performance improvement. The improvement from

---

14Larger values of $k$ are not discussed, because the amount of computation per input datum dominates the time taken to retrieve that datum, and so results in performance that is independent of the scheduling policy.
Figure 4.10: Execution time for the first 20 iterations of $k$-means clustering, using the default scheduler, sweetheart references, and in-memory caching.
the second iteration onwards is greater in the I/O-bound \((k = 1)\) case, for which sweetheart references alone reduce the iteration length by 52\% (Figure 4.10(c)) and the soft cache reduces the iteration length by 87\% (Figure 4.10(e)). The benefits in the moderately CPU-bound \((k = 100)\) case are more modest: sweetheart references reduce the iteration length by 16\% (Figure 4.10(d)), and the soft cache reduces the iteration length by 42\% (Figure 4.10(f)).

One criticism of this approach is that CIEL’s greedy scheduler does not identify optimal matchings between tasks and workers, even though a perfect matching—i.e. with all tasks data local—may exist. For example, with the two-random-choice placement policy, the probability of a perfect matching is approximately 58\%\(^{15}\). Therefore, with a sufficiently-advanced scheduler, it is more likely than not that all iterations could be performed with all tasks data-local, and certain that all tasks from the second iteration onwards would be data-local. For example, Isard et al.’s Quincy scheduler [IPC+09] uses a flow-based approach that would compute the optimal matching in this case. However, Quincy does not include the notion of a dynamically-changing preferred location for an object (i.e. a sweetheart reference), and it would be interesting future work to combine the two approaches.

### 4.4 Fault tolerance

In a large cluster built from inexpensive, commodity hardware, it is likely that some components will fail. Mainstream execution engines, such as MapReduce and Dryad, have become popular in part because they can transparently deal with hardware or network, without any need for programmer intervention. MapReduce and Dryad tolerate only one kind of failure: crash-failure of a worker machine [DG04, IBY+07]. An iterative CIEL computation, however, may run for much longer than a bounded-length MapReduce or Dryad job, which makes it more likely that any machine may fail, including those machines hosting the client and the master. This section describes and evaluates the mechanisms that are used to provide fault tolerance for any machine in a CIEL cluster: the client (§4.4.1), the workers (§4.4.2) and the master (§4.4.3).

#### 4.4.1 Client fault tolerance

In an iterative CIEL job, the involvement of the client is minimal: its only role is to upload any input data to the cluster and submit the job. After job submission, CIEL executes the whole job without involvement from the client.

By contrast, in a non-iterative framework, the client must run a driver program outside the cluster, in order to perform all data-dependent control flow. The structure of a driver program is given in Subsection 2.3.4. Because the driver program executes outside the cluster, it does

---

\(^{15}\)This figure was calculated by simulating the assignment of objects to workers, and solving the minimum-cost maximum-flow problem [Kle67] on a flow network derived from the bipartite graph formed by object assignments.
not benefit from the transparent fault tolerance that a distributed execution engine provides. Therefore, if the client machine fails or the network between the client and the cluster becomes partitioned, the driver program will fail. However, the driver program contains critical state: in a simple fixpoint computation, the execution stack will contain a pointer to the result of the latest iteration.

By providing an execution model that supports iteration (and fault tolerance as described in the following subsections), CIEL can execute an iterative computation with fault tolerance from beginning to end. Chapter 5 discusses programming models that allow a developer to write programs with a single logical thread of execution, in a similar style to a driver program.

### 4.4.2 Worker fault tolerance

As the most numerous participant in a CIEL cluster, the workers are the most likely source of failures. When a worker fails, it may cause two distinct failures:

1. Any tasks currently executing on that worker will fail.
2. Any objects stored on that worker will become unavailable.

The first class of failure is simple to handle: the master records the tasks that are currently assigned to each worker, and monitors whether each worker is still alive. While a worker is running, it periodically sends a heartbeat message to the master, which updates a timestamp recording the most recent successful heartbeat. In the master, a reaper thread periodically scans the worker pool to identify workers that have not sent a heartbeat in the past $t$ seconds. If the worker remains unresponsive after $t$ seconds have elapsed, the master considers that worker to have failed, and reschedules all of the tasks assigned to that worker.

If a temporary network partition causes heartbeat messages from a worker to be lost, and the worker eventually resumes contact, the master forces the worker to register again with the master. As part of the registration process, the worker sends a list of the concrete objects that it has stored to the master. Therefore, when the worker registers again, the master adds the worker’s objects to the dynamic task graph, which may cause some tasks to become runnable.

CIEL uses two strategies to deal with missing input objects, viz. replication and re-execution. An object may be replicated across many workers by storing it in several object stores, and recording the network locations of those stores in a concrete reference to the object. A worker attempting to read the object pointed to by a concrete reference will try the “nearest” location first, followed by the second-nearest, and so on. If none of the replicas is available, the consuming task will fail due to a missing input, and revert to the blocked state.

---

16The current implementation of CIEL only distinguishes between local and remote objects, and attempts to fetch from remote workers in a random order. However, it would be straightforward to use network topology information to prefer rack-local replicas [GGL03, IPC+09].
When a task fails due to a missing input object, the worker publishes a tombstone reference containing the network locations that should be invalidated because they no longer possess the object. When the tombstone reference is applied to the dynamic task graph, it removes those network locations from the set of replicas, and, if the replica set is empty, downgrades the object to a future. Table 4.4 shows the rules for updating the reference table when a tombstone reference is received. Note that the master does not immediately apply tombstone references to the dynamic task graph: it first investigates whether the relevant worker(s) have failed (by sending a heartbeat message to them). If a worker does not reply to a heartbeat, the tombstone reference is applied to the dynamic task graph, and the worker is treated as having failed completely. This additional check is necessary because the worker may have been experiencing temporary congestion, but the master has not deemed the worker to have failed. If this is the case, the tombstone reference should not be applied to the task graph, and the failed task will be rescheduled.

If the missing input object was produced by executing another task, CIEL can handle the failure by re-executing the task that produced that object. The master’s object table records the task that either produced or is expected to produce each object (§4.1.2). Therefore, a missing object can be recreated by re-running the lazy evaluation algorithm (§3.2.2) for that object, which will ultimately re-execute the task that produces the object.

### 4.4.3 Master fault tolerance

As the single component that coordinates parallel execution, the master must be online for a CIEL computation to make progress. Since CIEL supports iteration, the expected job lifespan is likely to be longer than for other execution engines: for example, when performing the same
iterative computation, the lifespan of a single CIEL job will be longer than the lifespan of each individual MapReduce or Dryad job. Therefore, because the CIEL master performs “more” coordination than in other frameworks, it is more important for the master to be fault tolerant. This subsection describes two techniques for master fault tolerance that have been developed for CIEL, but could equally be applied to less-powerful execution engines.

The state of a running job is fully encapsulated in the dynamic task graph, and hence in the task and object tables in the master. A trivial solution would be to record only the initial job submission message, which, since each task is by definition deterministic, would lead to the recreation of the whole task graph. Although memoisation would obviate the need to re-execute many data-intensive tasks, it would nevertheless be necessary to replay the control flow up to the point at which master failure occurred, in order to determine the object names (§4.2.1).

A better solution is possible by observing that only three operations modify the dynamic task graph: the initial job submission, task spawning and reference publication. Therefore, the task graph can be rebuilt by replaying these operations. This leads to two obvious strategies for master fault tolerance: recording the operations in a persistent log and mirroring the operations to a secondary master.

The persistent log approach (Figure 4.11(a)) creates one log file per job. When a job is submitted, a new log file is created and the initial log entry containing the job submission message is written synchronously to that file, before replying to the client. Any input files must also be written to durable storage, which may involve writing them to the same store as the log, or storing multiple replicas of each input (e.g. on several workers). This ensures that, if the client receives an acknowledgement from the master, the master has stored the minimum necessary information in order to replay the job (by the trivial argument above). Thereafter, all spawn and publish messages that the master receives can be written to the log asynchronously. After the master fails, a new master will replay the log, applying each operation in order to rebuild the
A UNIVERSAL EXECUTION ENGINE

dynamic task graph for the job. If the final log entry is truncated, the master discards it. Finally, the master restarts the job by lazily evaluating the output of the root task (i.e. the first log entry).

The secondary master approach (Figure 4.11(b)) is similar to the persistent log approach, except that the job submission message and all spawns and publish messages are forwarded to a secondary master. The secondary master immediately applies these operations to build a hot standby version of the dynamic task graph. To maintain the same reliability guarantees, the master must wait until the secondary master acknowledges the job submission message before returning an acknowledgement to the client; all other messages may be sent asynchronously.

Note that more elaborate logging architectures are possible. For example, since the behaviour of each operation on the task graph is deterministic, it would be possible to implement the master as a replicated state machine, which could easily be extended to provide Byzantine fault-tolerance [Sch90]. However, existing replicated state machine approaches are overly conservative, since they rely on all replicas receiving messages in the same order. There is no requirement for spawn and publish messages to be received in the same order, because each spawned task descriptor includes the name of its parent task: therefore the “maximal” task expected to produce each object can be computed from an unordered set of tasks (§3.1). This would permit a more decentralised form of logging, whereby a worker, upon completing a task, stores the spawn and publish messages locally, and sends them to a small number of randomly-selected hosts. This would be similar to relaxed-consistency update propagation in Grapevine [BLSN82], without the requirement that the log entries spread to all machines in the cluster. Recovering from master failures would then entail gathering partial logs from the workers to reassemble a complete log.

4.5 Summary

This chapter has presented the design and implementation of CIEL, which is a distributed execution engine that can execute a dynamic task graph as a single job. In particular, Sections 4.2, 4.3 and 4.4 showed how distributed storage, locality-aware scheduling and transparent fault tolerance—which are key features in a distributed execution engine—are implemented in CIEL. In addition, CIEL includes unique features that are particularly useful for long-running iterative computations: namely, a scheduling policy based on sweetheart references that improves performance with large loop-invariant data (§4.3.2), and fault tolerance mechanisms that allow a long-running computation to survive failure of the master (§4.4.3).

Up to this point, I have presented an execution model and a system that divides computation into abstract “tasks”, without describing how a task is implemented. The following chapter describes how various programming models can be built on top of CIEL, though a variety of executors.
Chapter 5

Parallel programming models

The great strength of the existing distributed execution engines is that they expose simple programming models: the programmer writes sequential code, which the system transforms into a parallel computation. By contrast, dynamic task graphs are very expressive but more difficult to specify, because the programmer is less constrained: each task can now choose whether to produce its outputs, spawn child tasks, or perform some combination of the two. This chapter shows how various programming models can be built on top of dynamic task graphs, in order to regain simplicity while allowing programmers to exploit the full expressiveness of the model.

In CIEL, support for different programming models is vested in the executors, which were defined abstractly in Chapter 3 as functions that transform tasks into new tasks and objects. In this chapter, I will make this notion concrete, by introducing several executors that provide different levels of functionality. The most basic executors, described in Section 5.1, provide no support for spawning tasks dynamically, but they may be used to implement existing static models, such as MapReduce [DG04] and Dryad [IBY+07].

As discussed in the earlier chapters, however, the most important feature of CIEL is the ability of tasks to spawn additional tasks dynamically. Section 5.2 introduces first-class executors, which give tasks the ability to spawn tasks programmatically through the executor interface. First-class tasks can be used to implement iterative and recursive parallel computations using a tail-recursive style of programming, which is similar to event-based programming: the disadvantage of this style is that the programmer must manually rewrite his algorithms to fit this style, by a process of “stack-ripping” [AHT+02]. Therefore, Section 5.3 demonstrates how a threaded programming model can be built on top of dynamic task graphs, by automatically transforming programs into a tail-recursive style using serialisable continuations.

5.1 Implementing existing models

In Section 2.3, I surveyed several data-flow-based systems that use a static, acyclic task graph to represent a computation. Compared to a dynamic task graph, a static task graph is relatively...
straightforward to program, because no task will spawn further tasks, and each task will produce all of its outputs. In this section, I will refer to such tasks as simple tasks. For example, in a task farm with independent tasks (§2.3.1), a MapReduce job (§2.3.2) or a Dryad job (§2.3.3), every task is a simple task. Although they are less powerful than dynamic task graphs, the widespread use of these systems illustrates that many computations can be carried out using only simple tasks. Therefore, in this section, I will discuss how CIEL supports simple tasks using a variety of simple executors.

The most primitive—but useful—simple executor is the shell executor (§5.1.1), which uses standard input and output streams to enable UNIX command-line utilities and other executable programs to be incorporated into an CIEL job. The simple Java executor (§5.1.2) is more flexible, because it uses Java stream objects to support multiple distinct inputs and outputs, and hence it can support MapReduce-style jobs. Finally, it is possible to implement iterative execution models (such as Pregel [MAB+10] and iterative MapReduce [BHBE10]), with a programming model in which user-defined tasks are simple tasks, and I discuss this possibility in §5.1.3.

5.1.1 Shell executor

Many UNIX command-line utilities are already designed to participate in a simple data flow, using inter-process pipes [RT74]. A pipe is a special file-like object, which is typically shared between two processes, where one process can write to the pipe and the other process can read from the pipe. The UNIX shell provides a convenient syntax for building pipelines of multiple processes—the following command

```
\texttt{detex thesis.tex} \mid \texttt{wc -w}
```

connects two processes in a pipeline: the first, \texttt{detex}, converts the \TeX document named \texttt{thesis.tex} to plain text, and the second, \texttt{wc}, counts the number of words in its input. The two processes run concurrently, and the pipe provides synchronisation\footnote{Additionally, the pipe provides flow control, because, in most implementations, \texttt{detex} will write into a bounded buffer, and will block if the buffer becomes full.}: \texttt{wc} blocks when it has consumed all of the available output from \texttt{detex}.

In CIEL, the synchronisation of a pipe can be simulated using data dependencies (§4.1.4) and flow control can be provided using streaming references (§4.2.3). Therefore, to allow UNIX-style utilities to execute as tasks, CIEL includes the shell executor, which exposes task inputs and outputs to those utilities as standard input and output streams.

The code dependency of a shell executor task is a key-value dictionary, containing the following information:
**Command line** A list of strings, which represent the argument vector for the process to be executed, including the name of the executable and any command-line arguments. This is a mandatory parameter.

**Inputs** A list of references, which comprise the data dependencies of the task. This is an optional parameter.

When the shell executor is invoked, it interprets the contents of the key-value dictionary. The executor forks a new process, and executes the given command line. The executor also creates a pipe between itself and the new process, which it assigns to the standard input file descriptor for the new process, in order to communicate with the process. The standard output of the new process is redirected to a temporary file. The executor then writes the entire contents of each input object, in the order specified in the inputs parameter, to the pipe. This is equivalent to invoking the `cat` utility [The08] where each argument corresponds to a file containing the respective object from the inputs parameter, and redirecting the standard output of `cat` to the standard input of the process. After the process exits successfully, the temporary output file is atomically moved into the local object store, using the name of the task’s expected output.

The shell executor supports fan-in (aggregation tree) parallelism, because it can consume multiple inputs, each of which may be produced in parallel. Therefore, it can be used to compute a simple variant of MapReduce, with a single reducer. However, because all output is collected from the task’s standard output, the shell executor only supports tasks with a single output, which means that it is not suitable for general MapReduce computations in which a map task may have multiple outputs\(^2\). In the following subsection, I will describe an executor that relaxes this restriction.

### 5.1.2 Simple Java executor

In the Dryad “vertex” programming model, each vertex (or task) is a sequential program that reads from zero or more input “channels” (or, in CIEL terms, objects), and writes to zero or more output channels [IBY+07]. To support Dryad-style computations, it was necessary to add an executor that supports tasks with multiple inputs and outputs: the simple Java executor.

The simple Java executor allows CIEL computations to be programmed in an object-oriented style, using the Java programming language. The programmer specifies the behaviour of a simple Java task by creating a class that implements the Task interface, which comprises a

---

\(^2\)Hadoop Streaming [Whi09, pp. 32–36] and Dryad (using the Nebula scripting language) [IBY+07] allow existing command-line utilities to be integrated into arbitrary data flows by enforcing record structure on the process output. By contrast, in CIEL, an object is an unstructured sequence of bytes (§3.1). It would be possible to construct an executor that interprets an input object as a sequence of records, and partitions those records between multiple outputs, but this chapter focuses on more-general executors that do not impose structure on task inputs and outputs.
interface Task {

    void invoke(InputStream[] inputs, OutputStream[] outputs, String[] args);

}

Listing 5.1: Simple Java tasks must implement the Task interface.

single invoke() method (Listing 5.1). A particular task can then be specified using a key-value dictionary, containing the following information:

Class  The fully-qualified name of a Java class that implements the Task interface. This is a mandatory parameter.

JAR libraries  A list of references, which comprise the set of Java archives containing the implementation of the necessary classes [Ora]. Each reference will also become one of the task’s data dependencies. This is a mandatory parameter.

Number of outputs  An integer, which must be greater than zero. This is a mandatory parameter.

Arguments  A list of strings, which comprise additional arguments to the task. This is an optional parameter.

Inputs  A list of references, which comprise the data inputs to the task. Each reference will also become one of the task’s data dependencies. This is an optional parameter.

When the simple Java executor is invoked, it instantiates a Java virtual machine (JVM), which dynamically loads the classes in the JAR libraries, instantiates the named class, and calls the invoke() method on the resulting object. The task inputs are provided as an array of InputStream objects, which may be backed by local files, in-memory buffers or network sockets; the executor makes the necessary objects available to the task. The task outputs are provided as an array of OutputStream objects, which are backed by temporary files; as with the shell executor, these are committed to the block store only after the task terminates successfully. Finally, additional arguments are provided as an array of strings, which are copied from the code dependency, allowing small parameters to be encapsulated in a single CIEL object.

Since Dryad is a generalisation of MapReduce, and the simple Java executor supports Dryad-style computations, the executor can also support MapReduce tasks. Figure 5.1 shows the pseudocode for a map task, implemented using the simple Java executor. The general strategy is to read each input record from the single data input, apply a user-defined map() function to a record, and write each emitted key-value pair to the appropriate output (determined by taking the hash of the key modulo the number of outputs). The corresponding reduce task would be
for all records, \(i\), in inputs[0] do
  for all emitted records, \(\langle k, v \rangle\), in map(i) do
    \(o \leftarrow \mathcal{H}(k) \mod |outputs|\)
    write \(\langle k, v \rangle\) to outputs[\(o\)]

Figure 5.1: Pseudocode for a naïve MapReduce map task implementation, using the simple Java executor.

a multiple-input/single-output task that merge sorts the inputs from different map tasks, and applies a `reduce()` function to each unique key and the associated values\(^3\).

The same principles in the simple Java executor can be applied to other languages. The .NET executor calls a method on a .NET class that implements a similar interface. The environment executor passes the names of input and output files to a process in environment variables, which allows this approach to be used in a language-independent manner, although more boilerplate code is required in this case.

5.1.3 Towards iterative data-flow

So far in this section, I have discussed how executors can be used to implement individual tasks. However, to obtain a dynamic task graph with many tasks, we still require a means of specifying the composition of those tasks. CIEL jobs contain a single root task, which is responsible for spawning the other tasks in the job, but this requires an executor that is capable of spawning tasks (unlike the executors defined in this section). Recall the implementation of MapReduce as a dynamic task graph from §3.3.1: the root task, \(t_{\text{root}}\), when executed, spawns tasks for the map phase and the reduce phase. The parameters of \(t_{\text{root}}\) are simple, and comprise a list of input references, a map task definition (\(t_M\)) and a reduce task definition (\(t_R\)). Both \(t_M\) and \(t_R\) can be described by a key-value dictionary, since they can be implemented as simple Java tasks (or equivalent). Therefore, \(t_{\text{root}}\) can also be represented as a key-value dictionary, and it would be possible to implement a MapReduce executor that interprets that dictionary, and spawns the appropriate tasks. The same approach could be used to construct an arbitrary Dryad computation, using a Dryad executor that interprets the domain-specific language for acyclic graphs introduced by Isard et al. [IBY\(^+\)07].

This approach can be extended to perform iterative computations. Recall the implementation of BSP/Pregel computations as a dynamic task graph from §3.3.2: we can add a Pregel executor that collects the votes from each vertex task in a superstep, and conditionally spawns another superstep if any task has not voted to terminate. As before, the Pregel tasks can all be specified using simple key-value dictionaries.

\(^3\)The scheme detailed here is deliberately naïve to simplify the exposition. A realistic MapReduce implementation would apply eager aggregation in the mapper, and perform buffering and preliminary sorting in the map task [YGI\(^+\)09]. CIEL includes a ported version of the Hadoop MapReduce implementation that provides these features.
However, this hypothetical proliferation of executors is unsatisfying, because it effectively requires an CIEL “plugin” to be written for each new programming model. The following sections describe an abstraction that allows a job to create arbitrary dynamic task graphs programmatically, and hence support a more flexible programming model.

5.2 First-class executors

A first-class executor is an executor that enables a running task to modify the control flow of a job by spawning additional tasks, delegating its outputs to those tasks and creating new objects programmatically. A first-class task is a task that runs on a first-class executor, and performs a sequence of spawn and publish operations that build up the task result (§5.2.1). In general, a first-class executor exposes an application programming interface (API) to the running task, which allows the task to perform these operations by invoking operations on the interface (§5.2.2). This API can be exposed to many different programming languages, and I describe the implementation for Java in Subsection 5.2.3.

5.2.1 First-class task semantics

Recall that the result of applying the executor function, \( E \), to a task, \( t \), is a subgraph, \( \Gamma_t \), comprising an object store and a set of tasks (§3.1, Definition 6). A first-class executor builds \( \Gamma_t \) by executing a sequence of \( N \geq 1 \) operations, \( \Omega_t = (\omega_1^t, \ldots, \omega_N^t) \), where:

\[
\begin{align*}
\omega_i^t &= \text{spawn} \ u \quad (u \in \text{Task}) \\
& \quad | \text{publish} \ n \mapsto o \quad (n \in \text{Name}, o \in \text{Obj})
\end{align*}
\]

Conceptually, the executor builds a sequence of subgraphs, \( (\Gamma_0^t, \ldots, \Gamma_N^t) \), where \( \Gamma_0^t \) is an empty graph, \( \Gamma_N^t \) is the final result, and each successive subgraph is constructed from the previous one.
by applying the corresponding operation. Figure 5.2 formally states the rules that are used to build the subgraphs: informally, a spawn operation adds a task to the current subgraph, and a publish operation adds a name-to-object mapping to the current subgraph.

This definition of a first-class task graph raises the valid concern that a cyclic dependency may be inserted in the graph. In general, one must perform a run-time check to ensure that the dependencies of each spawned task, \( u \), are already defined in either the global object store, or the previous local object store. Otherwise, it would be possible for deadlock to occur. The following subsection discusses a programming interface that constrains the spawn and publish operations to make this a static property of a first-class task.

5.2.2 Executor interface

The executor interface is a language-independent programming interface that first-class tasks can use to spawn tasks and produce objects. In addition to supporting the spawn and publish operations defined above, the executor interface is responsible for choosing the names for task outputs and new objects. It can therefore ensure that deadlock cannot occur.

The executor interface exposes a minimal set of operations that are required for data-dependent control flow:

\[
\text{construct}(o) \rightarrow n
\]

The construct operation creates a new object containing the data \( o \), and returns a uniquely-named concrete reference to that object.

\[
\text{spawn}(d_{\text{code}}) \rightarrow \langle n_1, \ldots, n_k \rangle
\]

The spawn operation spawns a new task with the given code dependency, \( d_{\text{code}} \), and returns a list of uniquely-named future references to the results of that task. The data dependencies and length of the result list must be computable from the code dependency.

\[
\text{publish}(\langle o_1, \ldots, o_k \rangle)
\]

The publish operation creates new objects containing the data in \( o_1, \ldots, o_k \), and maps those outputs respectively to the \( k \) expected outputs of the current task.

\[
\text{tail-spawn}(d_{\text{code}})
\]

The tail-spawn operation spawns a new task with the given code dependency, \( d_{\text{code}} \), and delegates the expected outputs of the current task to the spawned task. The current task must have the same number of expected outputs as the spawned task, which is computed from the code dependency (as for spawn, above).

A valid execution comprises any sequence of construct and spawn operations, followed by either a publish operation or a tail-spawn operation. The key property of this interface is the

\footnote{This interface was designed in collaboration with Christopher Smowton.}
fact that a task can only create an object name (concrete or future reference) by constructing an object or spawning a task. This follows inductively from the fact that a root task is, by definition, always runnable. Each subsequently-spawned (or tail-spawned) task may only depend on objects for which names have previously been created, i.e. objects that have previously been constructed (by a runnable task), or the results of a previously-spawned task. Therefore, it is not possible for a cyclic dependency to occur in the dynamic task graph.

The tail-spawn operation maintains the invariant that a task either produces all of its expected outputs, or delegates them to a child task (§3.2.1). Restricting tasks to all-or-nothing delegation does not reduce expressivity, because it is always possible to spawn multiple tasks that execute in parallel, and tail-spawn a task that depends on those tasks and copies their results to the delegated outputs of the parent task. However, this means that a task that depends on a subset of the task outputs will only execute when all of those outputs have been produced, which potentially introduces a synchronisation delay if that subset becomes available before the whole set of outputs. Selective delegation is possible in the dynamic task graph model, but the all-or-nothing model provides a simpler programming interface that is sufficient for the applications discussed in this dissertation.

In practice, it is desirable that the executor interface supports additional operations. For example, the implementation of the interface in CIEL provides a two-phase mechanism for object construction and output publication, which enables the task to produce objects using a file-like, streaming API—the executor interface exposes open and close operations for those streams, and an object only becomes concrete after a corresponding close. The executor interface also supports reading objects from the local or remote object stores, and provides an implementation of streaming between tasks (§4.2.3).

The current implementation of the executor interface uses two named pipes (one for requests from the task to the executor and another for responses) for control traffic, which uses a protocol based on JavaScript Object Notation (JSON) [Cro06]. More-efficient RPC representations and protocols are abundant, but JSON was particularly appealing because it is simple to parse and generate in many languages, including Java (§5.2.3), Python, C and OCaml, which has made it straightforward to create first-class executors for those languages.

5.2.3 First-class Java executor

The first-class Java (FCJava) executor is an extension of the simple Java executor (§5.1.2) that provides running tasks with access to the executor interface defined above. Similar to simple tasks, the programmer implements the behaviour of an FCJava task by writing a class that implements the FirstClassJavaTask interface (Listing 5.2). However, tasks may be invoked in two different ways:

---

5These names are created using the naming scheme presented in §4.2.1.

6However, it is possible that a task could never become runnable if the job itself diverges, for example by performing an infinite sequence of tail-spawn operations.
interface FirstClassJavaTask {

    // Called when the task is executed.
    // Externally-provided arguments are available in the Ciel.args array.
    void invoke();

    // Returns an array of Reference objects corresponding to the task’s
    // data dependencies.
    Reference[] getDependencies();
}

Listing 5.2: First-class Java tasks must implement the FirstClassJavaTask interface.

Externally-invoked task  The task arguments are a class name, and a list of string arguments, as for simple Java tasks (§5.1.2). This requires the entire behaviour of the task to be specified using strings, which can be accessed using the static Ciel.args array. However, it allows the bootstrapping of a first-class Java job from, for example, a plain-text key-value dictionary, such as the job submission package (§4.1.3).

Internally-invoked task  The task arguments include a reference to a serialisable functor object, which implements FirstClassJavaTask and encapsulates both the computation and the initial state of the task. This enables the behaviour of and inputs to a task to be specified using the full Java object model.

In general, the root task of a first-class Java job is an externally-invoked task, which then spawns one or more internally-invoked tasks. Using internally-invoked tasks allows the programmer to wrap task results and other references in statically-typed Java objects. For example, the FCJava library includes the generic SingleOutputTask<T> class, which allows the programmer to specify the task as a function of type T, and automatically publishes the return value of that function as the task result. The corresponding FutureReference<T> interface is used to represent the result of spawning a SingleOutputTask<T>.

Note that the invoke() method of FirstClassJavaTask does not include the task inputs or outputs as parameters. Instead, these are typically supplied as instance variables of the class that implements FirstClassJavaTask. The getDependencies() method enables the FCJava runtime to extract the data dependencies when building a task descriptor for an internally-invoked task.

Because it supports the tail-spawn operation, the FCJava executor is sufficiently powerful to express iterative and recursive algorithms. A data-parallel iterative algorithm can be implemented with three implementations of FirstClassJavaTask:
• An initial task that spawns \( n \) worker tasks, and tail-spawns a fixpoint task that depends on the worker tasks.

• A worker task that performs some partition of a data-parallel algorithm for the current iteration, and publishes the result as its output.

• A fixpoint task that reads the results of the \( n \) worker tasks, and evaluates whether they have converged to a fixed point. If they have not, the fixpoint task spawns \( n \) further worker tasks, and tail-spawns another fixpoint task that depends on the new worker tasks. If they have converged, the fixpoint task publishes the aggregated results of the worker tasks as its output.

For example, the worker tasks could compute PageRank updates for partitions of a large directed graph [PBMW99] or partial sums for \( k \)-means clustering (§6.4). Note that this task skeleton forms the same task graph as a Pregel-style computation (§3.3.2), and so any Bulk Synchronous Parallel algorithm can be implemented in this manner. Furthermore, the worker tasks need not all be identical, nor need they be independent: the ability to specify dependencies allows any MapReduce- or Dryad-style computation to be spawned dynamically.

In summary, the FCJava executor—via the executor interface—supports iteration and recursion by programming in a tail-recursive style, which is similar to the continuation-passing style (CPS) [SS98]. The use of CPS is enforced because a CIEL task may not block a worker indefinitely, and hence some portion of the environment—such as an iteration counter—must be passed to the tail-spawned task. However, CPS is not the most natural programming style for all iterative and recursive algorithms. The following section introduces an approach that automates the translation to CPS, and hence allows programming in a more-natural imperative or functional style.

### 5.3 Distributed thread programming model

The main disadvantage of the first-class-task programming model is that the programmer must identify all of the synchronisation points—i.e. any point in the computation where the result of an asynchronous task is required—and decompose the computation into separate tasks that execute before and after each synchronisation point. Any pre-synchronisation state that will be used later must be encapsulated in an object, and passed explicitly to subsequent tasks: this is equivalent to the stack-ripping problem in asynchronous and event-based programming, whereby any variables allocated on the stack are unavailable to subsequent callbacks or event handlers [AHT+02]. This problem arises because there has hitherto been a one-to-one mapping between application-level and system-level threads\(^7\). In this execution, I introduce the

---

\(^7\)Recall from §2.2.1 that a thread is a “single sequential flow of control”, which can be implemented with an instruction pointer and a private execution stack [Bir89].
Figure 5.3: Dynamic task graph for a distributed thread, comprising three tasks. The expected output of each task is the return value, $r$. The code dependency of each task is a continuation $(c_0, c_1, c_2)$, and tasks $t_1$ and $t_2$ have additional data dependencies on future objects $b_1$ and $b_2$, respectively.

The thread runtime decomposes a distributed thread into several tasks whenever it encounters an attempt to read a future object. When this occurs, the runtime will tail-spawn a new task that depends on that object and the current continuation of the task. Figure 5.3 shows the dynamic task graph of a distributed thread that blocks twice: first on future object $b_1$, then on future object $b_2$. As a result, the thread decomposes into three tasks $(t_0, t_1, t_2)$ which each have a code dependency on a different continuation $(c_0, c_1, c_2$, respectively). Each continuation represents the current execution state of the thread, at the point that it blocked: therefore, it contains the current snapshot of the thread-private data space, and a representation of the current execution state. The execution state may be represented in various forms, but it typically contains the contents of the execution stack between the thread entry point and the blocking function (Figure 5.4).

A programming language is suitable for distributed thread execution if it is possible to generate

\[\text{distributed thread programming model, which simplifies synchronisation by allowing a single application-level thread to extend across multiple tasks.}\]

In the threaded programming model, a computation is represented by a distributed thread, which is a single, sequential instruction stream that may execute in a chain of one or more tasks. Unlike shared-memory threads (§2.2.1), a distributed thread does not have write-access to a shared memory space. Instead, each distributed thread has read-write access to a thread-private data space, and read-create (but not modify) access to the CIEL object store.

A distributed thread is defined by a thread function, which may be implemented in any of the languages that are discussed later in this section. Each task in a distributed thread has the same expected output, which corresponds to the ultimate return value of the top-level thread function\(^8\): when the thread function returns, the return value is implicitly published as that output. As with any task, the act of spawning a distributed thread also creates a future reference to the thread’s expected output.

A programming language is suitable for distributed thread execution if it is possible to generate

\[\text{It is possible for a distributed thread to have multiple outputs, but the programming languages in this section do not provide convenient syntax for returning multiple values, which limits the usefulness of such a feature.}\]
function bar() {
    // Block on future object.
}

function foo() {
    bar();
}

function main() {
    foo();
}

Figure 5.4: When a distributed thread blocks, the continuation object includes a serialised representation of the stack frames between the thread entry point and the blocking method.

a serialised continuation. In the remainder of this section, I will discuss three approaches to this problem. In Subsection 5.3.1, I will introduce Skywriting, which is a scripting language for coordinating parallel execution. In Subsection 5.3.2, I will demonstrate how language-level techniques can be used to implement distributed thread support in the Scala programming language. Finally, in Subsection 5.3.3, I will show how OS-level checkpointing can be used to provide language-neutral support for threaded execution.

5.3.1 Skywriting

Skywriting is a dynamically-typed, interpreted scripting language, with a syntax that is based on C and JavaScript [ECM09]. The main role of Skywriting is to perform coordination between tasks written for other executors; therefore, the most important language features involve object management, task creation and task synchronisation.

To support a hybrid functional/imperative programming style, Skywriting allows assignment to variables in the thread-private data space, but all Skywriting functions are pure. Consequently, the body of a Skywriting function may assign to local variables, but it may not modify any variable that has been captured from an enclosing scope. This restriction enables Skywriting functions to run in parallel threads with the same semantics as a synchronously-evaluated function, and ensures that all Skywriting scripts are deterministic.

A variable may contain a value, a callable or a reference. The value types include integers, booleans, strings, lists and dictionaries, all of which may be constructed or interrogated in a Skywriting script. The callable types include named functions, anonymous functions and lambda expressions, all of which may be constructed or invoked. References correspond to objects in the Ciel object store, and therefore may not be constructed or interrogated directly. Instead, a reference is created using either the package() function, which transforms a job-private symbolic name (§4.1.3) into a reference, or the ref() function, which transforms a
function f(arg) {  
  return "Hello, " + arg + "!";
}

result = spawn(f, ["world"]);  
return *result;

Listing 5.3: Skywriting script that spawns a single task and blocks on the result.

words = spawn_exec("shell", {"inputs": [ref("http://example/thesis.tex")],  
    "command_line": ["detex"], 1});

count = spawn_exec("shell", {"inputs": [words[0]],  
    "command_line": ["wc", "-w"], 1});

return *count;

Listing 5.4: Skywriting script to perform \texttt{detex thesis\.tex | wc -w}.

URI into a reference. Note that CIEL objects are immutable: assigning to a variable containing  
a reference leaves the target object unchanged, and merely overwrites the reference. Although  
references may be dereferenced (see below), they may not appear on the left-hand side of an  
assignment expression, unlike dereferenced C pointers and C++ references [ISO03, §3.10].

Listing 5.3 illustrates the two most important features of Skywriting: \textit{spawning} and \textit{dereferencing}. The built-in \texttt{spawn()} function spawns a parallel thread that executes the given function  
with the given list of arguments: on line 5, a task will be created to compute \texttt{f("world")}. The  
value assigned to \texttt{result} is a future reference, which corresponds to the expected output  
of the spawned task. On line 6, the unary-* (dereference) operator is applied to \texttt{result}, which  
blocks the current thread until the future reference has become concrete, and then yields its  
value—"Hello, world!" in this example.

Although simple calculations can be made in Skywriting, it is principally used to spawn tasks  
written in other languages. Listing 5.4 shows a more sophisticated Skywriting script, which creates  
a two-task pipeline to perform the shell command \texttt{detex thesis\.tex | wc -w}. The \texttt{spawn_exec()} function is used to spawn tasks that are written in languages other than Skywriting: the first argument is the name of the executor that will execute the task and the second argument is a key-value dictionary of arguments. The third argument is the number of expected outputs, and the return value is a list of the same number of future references, corresponding to those outputs. The listing uses \texttt{spawn_exec()} to create two tasks that use the shell executor  
(§5.1.1) to execute \texttt{detex} and \texttt{wc}, respectively.

Listing 5.4 also demonstrates how future references enable data flow between tasks: the out-
function fib(n) {
  if (n <= 1) {
    return n;
  } else {
    x = spawn(fib, [n - 1]);
    y = spawn(fib, [n - 2]);
    return *x + *y;
  }
}

return fib(10);

Listing 5.5: Skywriting script for computing the 10th Fibonacci number

put of the first spawned task is passed as an input to the second. Since Skywriting futures are first-class, they may be passed to another task without blocking on their values: this enables Skywriting scripts to build up arbitrary acyclic dependency graphs\(^9\). Futures prevent also deadlock in Skywriting, because the arguments to a task are passed by value, and the corresponding futures are created only after the arguments have been passed. Therefore, a task’s arguments cannot contain its own future, nor can they contain a future from a subsequently-spawned task. As a result, Skywriting inherently prevents the programmer from writing a script that contains a cyclic dependency, which would lead to deadlock.

Note that the dereference operator is necessary in order to enable data-dependent control flow: it allows a Skywriting thread to access the result of a parallel task, and base subsequent control flow decisions on that result. If the parallel task is part of a Skywriting thread, the result may be any valid Skywriting value or reference. However, if the parallel task is implemented using a different executor, the two executors must agree on a common data representation for the object that passes between them. The current implementation of the dereference operator assumes that any dereferenced object is stored in JSON [Cro06], which enables tasks to return strings, numbers, booleans, and (possibly-nested) lists and key-value dictionaries. Note that the use of JSON is mandated only for task results that will be dereferenced: Skywriting is primarily designed to coordinate non-JSON data flow between tasks that are implemented using other executors.

A spawned Skywriting function is also a first-class distributed thread: it may also spawn tasks or threads, and dereference task results. Listing 5.5 shows a Skywriting script that recursively calculates the 10th Fibonacci number. This example illustrates how Skywriting can be used

---

\(^9\) This can be contrasted with the implementation of `spawn` in Cilk [FLR98]. In Cilk, applying `spawn` to a function-call expression does not change the type of the expression. This ensures that the serial elision of a Cilk program is a valid C program, but it prevents the unevaluated expression from being returned or passed to another `spawn`. Hence Cilk supports only fully-strict or fork-join parallelism, whereas Skywriting additionally supports dependencies between two tasks spawned by the same parent.
to define a data-dependent parallel algorithm\(^\text{10}\). For \(n > 1\), the \(\text{fib}(n)\) function spawns two threads to calculate \(\text{fib}(n - 1)\) and \(\text{fib}(n - 2)\), then dereferences the results of these tasks, adds them together, and returns them. The spawned threads may themselves also spawn threads and dereference their results. Although I do not consider such algorithms in this dissertation, this example suggests the possibility of using Skywriting and CIEL to execute parallel divide-and-conquer algorithms, such as decision tree learning [Qui86].

**Skywriting script execution**

The Skywriting language is implemented as a first-class executor that includes an interpreter for Skywriting scripts. The key idea is that each task includes a continuation object as one of its dependencies. I will now briefly discuss how the Skywriting executor translates script execution into a dynamic task graph.

The first task in a Skywriting job depends on a single object that contains a plain-text representation of a Skywriting script; this approach can also be used to spawn a Skywriting thread from another non-Skywriting task. The executor must therefore parse the script to generate an abstract syntax tree, and construct an empty thread-private data space. By contrast, the initial task in a spawned thread depends on a **closure**, which contains a serialised representation of the function to be executed, and the values of any parameters and free variables. In both cases, the executor then begins executing the script or closure with an empty execution stack.

The interpreter proceeds by traversing the AST, starting with the top level, which is a list of statements. On encountering an assignment statement, the interpreter evaluates the expression on the right-hand side, and updates the thread-private data space, by binding the storage location on the left-hand side to that value. Variable names are lexically-scoped; however, a statement may only assign to a variable if it is declared in the same context (i.e. top-level or function body) as the statement. Control-flow statements, such as **while**-loops and **if**-statements are implemented by repeatedly or selectively traversing subtrees of the AST.

Most expression evaluation is side-effect-free: however, evaluating the built-in **spawn()** and **spawn_exec()** functions will invoke the **construct** and **spawn** operations on the executor interface (§5.2.2). Task spawning follows the same structure as other first-class executors: first, the script constructs an object that corresponds to the task’s code dependency, then it spawns a task that depends on that object. Finally, the interpreter stores the names of the task outputs in the local environment. Figure 5.5(a) shows the dynamic task graph fragment arising from a call to **spawn()**. In this case, the spawned task depends on a **spawned function**, which contains the AST of callable argument to **spawn()**, the list of actual parameters, and any captured variables. Figure 5.5(b) shows the corresponding graph for a **spawn_exec()** task, which depends on a **parameter object** that contains a serialised copy of the task arguments, and any data

\(^{10}\)It is customary to note that this is an inefficient method of calculating Fibonacci numbers [FLR98]. Nevertheless, it serves as a useful demonstration of Skywriting language features.
**CHAPTER 5. PARALLEL PROGRAMMING MODELS**

**Figure 5.5:** Dynamic task graphs for (a) the initial task in a job or spawned thread, and (b) a non-Skywriting task created with `spawn_exec()`.

**Figure 5.6:** Dynamic task graph resulting from a Skywriting script that spawns two tasks and blocks on their results.
dependencies\textsuperscript{11}.

As described above, the dereference operator blocks a Skywriting thread by tail-spawning a task that depends on the blocking references\textsuperscript{12}. To achieve this, the executor constructs an object containing the current execution state—the AST, the thread-private data space and the execution stack; collectively, the \textit{continuation} of the task—and sets the code dependency of the tail-spawned task to that object. Figure 5.6 shows the dynamic task graph that results when a Skywriting task spawns two subtasks and dereferences their results. The AST and thread-private data space are straightforwardly serialised; however, in a conventional interpreter, the execution stack is stored on the runtime stack of the interpreter, which is not usually serialisable. Therefore, the Skywriting interpreter maintains the execution stack as an application-level data structure, by pushing a frame for each AST node as the interpreter visits it. This is similar to the approach of Stackless Python [Tis00], which uses a chain of Python objects to represent the execution stack\textsuperscript{13}.

This subsection has demonstrated how Skywriting enables a threaded (imperative or functional) programming model on top of CIEL. The key observation is that the ability to serialise a portable continuation enables a single thread (that may block) to be decomposed into a chain of tasks (that may not block). However, Skywriting is a domain-specific interpreted language, which lacks the efficiency and the library support needed to implement data- or CPU-intensive computations. As a result, the bulk of a job must usually execute in tasks created by \texttt{spawn\_exec()}. The following subsection investigates how the concepts introduced by Skywriting can be implemented in a compiled language.

### 5.3.2 Scala

Scala is a “multi-paradigm” programming language that integrates concepts from object-oriented and functional programming. It is statically-typed and compiles to Java Virtual Machine byte-code, which enables it to interoperate with classes written in Java [OAC\textsuperscript{+}06]. However, Scala is particularly interesting because its compiler includes support for \textit{delimited continuations}, due to the work of Rompf \textit{et al.} [RMO09]. Whereas a general continuation represents the entire remainder of a computation, a delimited continuation is bounded by a programmer-specified \textit{delimiter} on the stack [DF90, Fel88]; this makes it possible to represent the delimited continuation using a serialisable object, which can therefore be stored as a CIEL object. As a result, it is possible to use Scala’s delimited continuations to implement distributed threads using the

\textsuperscript{11}When a task is spawned, the target executor is invoked to build the descriptor for the new task. This allows the target executor to verify the arguments, and extract any additional dependencies.

\textsuperscript{12}Multiple references are supported by lazily evaluating the dereference operator. The initial application of the dereference operator creates a thunk. The task blocks on all dereferenced references at the first point when the value of a thunk is required (for example, in an arithmetic expression).

\textsuperscript{13}Christopher Smowton has subsequently implemented the “SkyPy” executor, which allows distributed threads to be written using the Python programming language.
```scala
var savedCont : (Unit => Unit) = null
println("0. Before reset.")
reset {
  println("1. Before shift.")
  shift { cont : (Unit => Unit) =>
    println("2. Inside shift.")
    cont()
    savedCont = cont
    println("3. End of shift.")
  }
  println("4. After shift.")
}
println("5. After reset.")
savedCont() // Prints the following:
0. Before reset.
1. Before shift.
2. Inside shift.
4. After shift.
3. End of shift.
5. After reset.
4. After shift.
```

Listing 5.6: Example of the control flow when using the `reset` and `shift` operations in Scala. Calling `cont()` inside the `shift` block invokes the code between the end of the `shift` block and the end of the enclosing `reset` block.

first-class Java executor (§5.2.3). This subsection describes how Scala enables the programmer to write both the coordination and computational code for a CIEL job in a single, efficient, statically-typed language.

Rompf et al.’s implementation of delimited continuations provides the primitive `reset` and `shift` operations described by Danvy and Filinski [DF90]. Listing 5.6 shows a code fragment that uses those operations to produce a delimited continuation, `cont`, and invoke it multiple times. The control flow between statements is also shown beside the listing. Entering the `reset` block pushes a delimiter onto the stack to indicate the limit of the continuation. Entering the `shift` block binds the current delimited continuation to an identifier (`cont`), which represents a `Unit` (void) function. When invoked, `cont` executes the code between the end of the `shift` block and the end of the enclosing `reset` block. In addition, within that scope, `cont` is a first-class value, so it may be assigned to a variable: in the example, `cont` is assigned to `savedCont`,
try {
    reset {
        // Invoke thread run() method.
        // Publish the return value of the run() method.
    }
} catch {
    case be: BlockException => {
        val continuation = be.cont
        val blockingRef = be.ref
        // Thread has blocked, so tail-spawn a continuation that
        // depends on blockingRef.
    }
}

Listing 5.7: The Scala thread entry point uses reset to delimit the continuation

class Future[T] extends Reference {
    ...
    def get : T @suspendable = {
        if (/* reference is not concrete */) {
            shift { (cont : Unit => Unit) =>
                throw new BlockException(cont, futureRef)
            }
        }
        // The object is now concrete, so attempt to open it through
        // the executor interface.
    }
    // Deserialise and return the object.
}

Listing 5.8: The Scala Future[T].get method uses shift to capture the continuation

which allows it to be invoked outside the reset block.

The Scala runtime for CIEL uses a combination of delimited continuations and exceptions to implement blocking on futures. Listing 5.7 shows pseudocode for the thread entry point, which uses reset to delimit the continuation at the start of the thread. Listing 5.8 shows the pseudocode for what happens when the thread attempts to dereference a Scala future: if the attempt fails because the corresponding object has not yet been produced, the shift block captures the current delimited continuation, and throws an exception containing the continuation (cont) and the blocking reference (futureRef), which is caught by the catch block in Listing 5.7. In this case, the exception handler creates a continuation task, which is a first-class Java task that treats the thrown continuation as a functor object, and additionally depends on the blocking
class ScalaThreadExample extends DistributedThread[Int] {
    override def run = {
        val x : Int = 15
        val y : Int = 73

        val first : Future[Int] = Ciel.spawnThread { _ => x * 2 }
        val second : Future[Int] = Ciel.spawnThread { _ => y + 7 }

        val result : Int = first.get + second.get
        // The last expression in a thread is the thread result.
        result
    }
}

Listing 5.9: Example of Scala threads, showing static types

class ScalaThreadExample extends DistributedThread[Int] {
    override def run = {
        val x = 15
        val y = 73

        val first = Ciel.spawnThread { _ => x * 2 }
        val second = Ciel.spawnThread { _ => y + 7 }

        first.get + second.get
    }
}

Listing 5.10: Example of Scala threads, using type inference

reference.

On becoming unblocked, the continuation task invokes cont, and continues from the point at which it blocked—i.e. after the shift block in Listing 5.8. At this point, the dereferenced object is guaranteed to be available (because it is a data dependency of the continuation task), so it is now possible to read the object, and the runtime deserialises it and returns it to the caller.

In the Scala implementation of distributed threads, the thread result is statically typed. The thread function has the type Unit → T, where T is a generic type parameter. The runtime exposes a function spawnThread, which has type (Unit → T) → Future[T]. A Future[T] object wraps a CIEL future reference, and implements the dereferencing get method shown in Listing 5.8. These definitions combine to enable static type-checking of thread
class ScalaGeneratorExample extends DistributedThread[Int] {

  override def run = {

    val generator = Ciel.spawnGenerator[Int] { _yield =>
      _yield(6)
      _yield(3)
      _yield(7)
    }

    val iter = generator.iterator
    val total = iter.sum

    // Result will be 16.
    total
  }
}

Listing 5.11: Example of Scala generator tasks

results. For example, Listing 5.9 shows an example Scala thread that spawns two child threads and blocks on their results. The child thread functions have type Unit → Int, so the result of spawning each thread has type Future[Int]. In Listing 5.9, all of the types are shown explicitly, but Scala’s type inference makes it possible to elide these, as shown in Listing 5.10.

The rich feature-set of Scala makes it possible to define other programming models, in addition to distributed threads. Listing 5.11 shows how to build a statically-typed generator task, which generates a stream of integers. The spawnGenerator function has type ((T → Unit) → Unit) → Generator[T]. The definition of generator makes this clearer: the generator is a function that has no return value and one parameter (_yield). The _yield parameter is a function that takes the generated elements (integers in this example), and writes them to the task output. In the parent thread, the generator exposes a (blocking) method that obtains an Iterator[T], which then can be used with built-in collective operations, such as sum in this example. The distributed Scala runtime makes it straightforward to implement higher-order operators—such as map() and fold()—and hence provide a programming model that is similar to DryadLINQ [YIF+08], Spark [ZCF+10] or FlumeJava [CRP+10].

A spawned Scala thread is a first-class distributed thread, in the same vein as the Skywriting threads described in §5.3.1. Listing 5.12 shows the Fibonacci example (cf. Listing 5.5) expressed in Scala. The principal advantage of this program over the Skywriting version is that it is statically-typed, which allows the compiler to detect, at compile-time, a class of bugs that would cause errors in the Skywriting script at run-time. Note that the type of the fib function is Int → Int @suspendable: the @suspendable annotation indicates to the compiler that this method should be transformed into continuation-passing style, because it uses
class Fibonacci extends DistributedThread[Int] {
  def fib(n : Int) : Int @suspendable = {
    if (n <= 1) {
      n
    } else {
      val first = Ciel.spawnThread { _ => fib(n - 1) }
      val second = Ciel.spawnThread { _ => fib(n - 2) }
      first.get + second.get
    }
  }

  override def run = {
    fib(10)
  }
}

Listing 5.12: Scala program for computing the 10th Fibonacci number

14 All methods on the execution stack between the thread entry point and a @suspendable method must also be annotated with @suspendable. One consequence is that it is not possible to use Future.get within a standard-library callback—e.g. when mapping over a collection—because most library methods are not annotated with @suspendable.
5.3.3 Process checkpointing

So far, all of the executors that support a threaded programming model require the program to be written in a specific language. This subsection investigates an approach that enables a multi-threaded process to run as a CIEL job without enforcing a particular programming language or style, by instead modifying the operating system.

Recall that the key idea of distributed threads is that, instead of blocking, the current task stores its execution state as an object, and spawns a new task that depends on both the execution state and the blocking object. To achieve this in a language-independent way, one can use checkpoint/restart, implemented as part of the operating system, to serialise the address space of the current process to disk. The Berkeley Lab Checkpoint/Restart (BLCR) system implements this facility for Linux, and it has been integrated with CIEL to create a checkpoint executor.

Listing 5.13 continues the running example of computing the 10th Fibonacci number, this time with a C program that uses the checkpoint executor. Although this example is written in C, any language with bindings to the executor interface can in principle use the checkpoint executor, since BLCR can checkpoint an arbitrary Linux process. The C bindings to the checkpoint executor are designed to resemble the POSIX thread management API [The08, §2.9], and they map onto the spawn and tail-spawn operations in the executor interface (§5.2.2).

When `ciel_thread_create()` is called, the runtime logically forks the current process, by saving a checkpoint of the process to create a copy of its current address space: this ensures that the thread function pointer and the `thread_data` pointer argument are valid in the new thread. The `ciel_thread_join_all()` function joins one or more spawned tasks, and makes their return values available to the parent. To achieve this, it transparently checkpoints the current task, and tail-spawns a continuation, as in the other threaded executors. It should be noted that the `ciel_thread_t` objects are more similar to first-class futures than child process IDs, so it is possible to pass them between tasks and hence a task can block on the result of a sibling task. The main task result is indicated by the value in the thread function’s `return` statement, but the C bindings also provide the ability to create multiple streaming outputs and/or deterministically modify shared data structures by using a merge policy based on the thread joining mechanism proposed by Aviram et al. [AWHF10].

Due to the limitations of the C type system, the checkpoint executor uses a simple tagged-object approach to pass return values between threads: the `ciel_make_int()` and `ciel_get_int()` functions respectively construct and interrogate an object that contains an integer. In order to allow the client and other executors to consume the results of a checkpoint-based job, the root thread uses standard output to write its outputs, in a similar manner to the shell executor (§5.1.1).

The principal limitation of this approach is that the checkpoint files contain the whole address space, which means that they are typically large (on the order of 8 MB per checkpoint, for a job

---

15 The implementation of process checkpointing for CIEL jobs was carried out by Sebastian Hollington for his Part II individual project [Hol11].
Listing 5.13: Checkpoint-based C program for computing the 10th Fibonacci number

ciel_val_t *fib(void *thread_data) {
    int n = *(int *) thread_data;
    if (n <= 1) {
        return ciel_make_int(n);
    } else {
        // Spawn fib(n - 1).
        --n;
        ciel_thread_t first = ciel_thread_create(fib, &n);
        // Spawn fib(n - 2).
        --n;
        ciel_thread_t second = ciel_thread_create(fib, &n);
        // Blocks the current thread and resumes when results are
        // available.
        ciel_thread_join_all(2, first, second);
        // Build a result from the two child thread results.
        int result = ciel_get_int(first) + ciel_get_int(second);
        return ciel_make_int(result);
    }
}

int main(int argc, char *argv[]) {
    fib(10);
    // The root thread writes outputs to standard output.
    printf("%d", fib(10));
}

that does not use the heap), and inefficient to move around a cluster. Since spawning a task also
involves making a checkpoint, this increases the latency of task creation. In addition, BLCR
imposes various restrictions on the behaviour of a checkpointable process: sockets will not be
preserved across a restart, and open files will only be preserved if the same file is available on
the destination machine [DHR02]. If the task uses only the executor interface, it avoids these
limitations, but the current checkpoint executor will run arbitrary binary code that it does not
attempt to sandbox. In order to provide this facility more safely, an isolation technique such
as Capsicum [WALK10], Xax [DEHL08] or Native Client [YSD+10] could be used to prevent
tasks from opening arbitrary files or sockets.
### Table 5.1: Rules for combining an existing fixed reference with an incoming tombstone reference.

<table>
<thead>
<tr>
<th>Existing ref.</th>
<th>Incoming reference</th>
<th>Tombstone,(_y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed,(_x)</td>
<td>Future</td>
<td>(if (x = y))</td>
</tr>
<tr>
<td>Fixed,(_x)</td>
<td>Fixed,(_x)</td>
<td>(otherwise)</td>
</tr>
</tbody>
</table>

A fixed reference, Fixed,\(_x\), has a single location \(x\).

#### Allowing tasks to block

In this section, I have strongly suggested that the ability to serialise the execution state is necessary for threaded execution. However, this is clearly not true, because non-iterative frameworks can simulate data-dependent control flow using a driver program that runs outside the cluster, submits jobs to the cluster, and blocks while the cluster executes distributed jobs. I have already critiqued this model in §2.3.4. However, this model has the advantage that it can work with an arbitrary legacy process, with no constraints on the programming language or system features that it uses. Therefore, CIEL also supports the legacy process executor, which uses the executor interface to spawn tasks and construct objects, but also provides the following additional operation:

\[ \text{block-on}(\langle d_1, \ldots, d_n \rangle) \]

The block-on operation tail-spawns a new task with data dependencies on the objects named \(d_1, \ldots, d_n\). The code dependency is a fixed reference, which is associated with the current worker, and contains a unique key that is derived from the current process ID and a serial number. Upon invoking this operation, the current process is blocked until all of the given data dependencies become concrete.

The fixed reference is an additional type of reference, in addition to those described in the previous chapter. A fixed reference has a single fixed location, which is almost always associated with the worker at which it was created. Like other concrete references, a fixed reference corresponds to an object, but that object can only be read locally at the fixed location: an attempt to read its contents from a remote object store will fail. Therefore, if the scheduler (§4.3) encounters a task that depends on a fixed reference, it must assign the task only to the fixed location for that reference\(^{16}\). Finally, Table 5.1 shows the rules for combining a fixed reference with an incoming tombstone reference in the event of a worker failure (§4.4.2).

The block-on operation makes a synchronous request to the executor, and blocks until a response is received. During that time, the process is in a blocked state: specifically, it is blocked

\(^{16}\)It is an error for a task to depend on two fixed references that have different fixed locations. The current executor interface does not permit fixed references to be constructed directly, which prevents this situation from arising. However, if such a situation should arise, the scheduler will publish error references for each of the task’s outputs.
on the `read()` system call on the executor interface response pipe. Upon receiving the request, the executor stores the current process and the file descriptors for the executor interface pipes in a worker data structure called the process pool. The process pool is a key-value dictionary, mapping UUIDs to a structure containing file descriptors for the executor interface FIFOs for each process. The process pool UUID for the blocked process is stored locally, and a fixed reference is manufactured to refer to this information. The executor effectively tail-spawns a task that depends on the given data dependencies, with the fixed reference as the code dependency to ensure that the continuation is scheduled on the same worker. When the task is finally dispatched to the same worker, the legacy process executor consults the code dependency to obtain the appropriate blocked process, and sends a response to the process in order to unblock it and continue execution. Note the identity of the current task changes during the period when a legacy process is blocked, which ensures that tasks spawned and objects produced are associated with the appropriate phase of execution.

Although a legacy process is pinned to a single worker, it can still enjoy transparent fault tolerance, as long as it is deterministic. If the worker hosting a legacy process should fail, CIEL’s fault tolerance mechanisms (§4.4.2) will recreate the state that corresponds to the latest fixed reference by re-executing the process from the beginning. Assuming that the legacy process mainly performs task spawning and other control flow, this will not be particularly time-consuming.

## 5.4 Summary

<table>
<thead>
<tr>
<th>Executor</th>
<th>Shell §5.1.1</th>
<th>Simple Java §5.1.2</th>
<th>FCJava §5.2.3</th>
<th>Skywriting §5.3.1</th>
<th>Scala §5.3.2</th>
<th>Checkpoint §5.3.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publish result</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Multiple results</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Spawn tasks</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Construct objects</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Distributed threads</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Statically typed</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Lang. independent</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
</tbody>
</table>

CIEL was designed to support a plurality of programming models, because there is no one programming model that is ideal for all jobs. Many of the example applications that I have developed for CIEL primarily use simple executors like the shell and simple Java executors, because they are adequate for simple tasks that stream over a large data object to perform MapReduce-style computations. However, the first-class executors become necessary for any job that contains data-dependent control flow: the first-class Java executor provides a tail-recursive programming model for iteration. Furthermore, the Skywriting, Scala and checkpoint executors allow developers to write code that spawns and synchronises with other tasks, while maintaining the illusion of a single thread of control. The approach generalises to any language
in which it is possible to capture the current execution state in a portable manner. The availability of delimited continuations for other functional languages—such as Haskell [DJS07] and OCaml [Kis10]—and the implementation of async tasks in F♯ [SPL11] suggest that many other languages could support a similar programming model.

Fundamentally, all of these programming models are variations on the spawn-and-publish execution model for dynamic task graphs that I outlined in Chapter 3. By developing several different programming models that cover a wide range of applications, including existing data-intensive models and general threaded programs, this chapter has demonstrated the practical expressiveness and utility of dynamic task graphs.
Chapter 6

Evaluation

Parallel programming is at least as difficult as its sequential counterpart, and usually more so [Gil58]. Therefore, a parallel execution engine such as CIEL is only practical if it provides some additional utility. Performance measures are a practical way of quantifying this utility, and in this chapter I will discuss the performance of CIEL in relation to other parallel frameworks. There are many different ways of quantifying performance, and I will focus on the following measures in this chapter:

**Speedup** Parallel execution often yields an increase in performance relative to sequential execution. The ratio of sequential execution time to parallel execution time is called the *parallel speedup*. Gene Amdahl formulated Amdahl’s Law, which states that the maximum parallel speedup for a given computation is limited by the sequential portion of that computation [Amd67]; therefore an efficient execution engine will limit the amount of time spent in sequential framework code. John Gustafson later observed that applying the same algorithm to a larger problem will tend to increase the amount of parallelisable work [Gus88]. In the age of “big data” [MCB+11], it would therefore be tempting to downplay the role of Amdahl’s Law. However, this would ignore the fact that some problems have a fixed size, and the ideal execution engine would deliver useful speedup on both small and large data sets. Therefore, a secondary metric is the range of problem sizes for which a “reasonable” speedup can be achieved.

**Efficiency** Adding more processing resources to a computation incurs a cost. This may be an opportunity cost, by depriving some other computation of execution time, or it may be a direct marginal cost if computation is an elastic resource, for example when purchasing virtual machines from a cloud computing provider [AFG+10, YBS08]. The ratio of parallel speedup to the number of processing elements is called the *parallel efficiency*. Since most algorithms exhibit parallel speedup that is sub-linear in the number of processing elements, the optimal efficiency is typically achieved in the sequential case, so parallel efficiency is typically in tension with absolute performance.
Absolute performance Parallel speedup and efficiency are ratios, which makes them useful for comparing two different configurations of the same system. However, the user’s primary concern is often absolute performance (or absolute cost), measured in seconds (pounds sterling) across the makespan of a computation. An unintuitive consequence of Amdahl’s Law is that improving the absolute performance of the parallel part of a computation will lead to poorer parallel speedup, because the relative weight of the sequential portion will increase. As a result, while reimplementing a fixed-size computation in a more efficient manner may yield a “less scalable” solution in terms of speedup, it will have better absolute performance, and therefore greater utility to the end-user.

In focusing on performance, one must not forget other aspects of system utility, which may be qualitative or only indirectly quantitative. For many computations, the optimally-efficient implementation would entail building application-specific hardware interconnected by highly-reliable network links. A concrete example of this approach is the Quantum ChromoDynamics On a Chip (QCDOC) supercomputer [BCC+04], which cost $1.6 million for a machine comprising 512 application-specific processors that are connected in a six-dimensional torus, and which is used for simulations of quantum physics. While custom hardware may give the best performance, and it may be affordable for some large institutions, it is diametrically opposite the commoditisation of parallel computing through distributed execution engines such as MapReduce, Dryad and Ciel. These systems exploit the economies of scale in commodity hardware production to scale incrementally: the marginal cost of increasing capacity is merely the cost of one more commodity server to add to the cluster. Finally, distributed execution engines support high-level programming models that are more straightforward for the programmer, because he need not, for example, write explicit synchronisation code. As the results in this chapter show, these high-level abstractions can negatively affect performance in some cases, but this may be tolerable if the increase in programmer productivity offsets the performance impact.

The remainder of this chapter presents the results of several performance evaluations, carried out using Ciel and other parallel frameworks. Section 6.1 describes the experimental configuration. The first set of experiments explores the performance of primitive task-handling operations in Ciel, in order to establish limits to its scalability (Section 6.2). The second set of experiments compares the performance of Ciel to the Apache Hadoop implementation of MapReduce when running MapReduce-style computations (Section 6.3). The third set of experiments compares the performance of Ciel and Hadoop when running the iterative k-means algorithm, and demonstrates the advantages of representing iteration within the execution engine (Section 6.4). The fourth set of experiments measures the cost and benefits of fault tolerance on a representative iterative algorithm (Section 6.5). Finally, the fifth set of experiments shows how the ability to stream data between tasks enables Ciel to achieve high data transfer throughput and parallel speedup in algorithms that have fine-grained data dependencies (Section 6.6).
CHAPTER 6. EVALUATION

6.1 Experimental configuration

All of the experiments in this chapter have been carried out using m1.small virtual machines (instances) leased from the Amazon Elastic Compute Cloud (EC2) [AWSa]. At the time of writing, an m1.small instance has the following specifications [AWSb]:

- 1.7 GB of RAM
- One EC2 Compute Unit in one 32-bit virtual CPU. An EC2 Compute Unit is equivalent to a 1.0–1.2 GHz 2007-model AMD Opteron or Intel Xeon processor.
- 160 GB of secondary storage split between a 10 GB root partition and a 150 GB data partition. In each configuration, the execution engine is installed on the root partition, and the application data are stored on the data partition.
- Moderate I/O performance, with I/O bandwidth proportionately shared between other virtual machines on the same physical host. Zaharia et al. showed that this can potentially lead to performance variation for heavy I/O workloads [ZKJ+08]; hence I performed additional repeated runs to compensate for this effect.

All instances were configured with a Linux 2.6.35-302-ec2 kernel, packaged in the Ubuntu 10.04 (Lucid Lynx) distribution\(^1\). The Hadoop experiments were carried out using version 3u0 of the Cloudera Distribution for Hadoop, which includes a patched version of Apache Hadoop 0.20.2. The MPI experiments were carried out using OpenMPI 1.4.1.

6.2 Task handling

The performance of CIEL’s task handling mechanisms determines the minimum granularity of computation that can execute efficiently as a CIEL job. The task is the smallest unit of computation that can be scheduled in a CIEL job. This fact has an impact on coordination in a CIEL job: in order to establish unidirectional communication from worker A to worker B, it is necessary to create a task that will run on B and consume an object that resides on A, or vice versa. Furthermore, because a dynamic task graph is acyclic, in order to establish bidirectional communication between the workers, it is necessary to create two tasks—or, for multidirectional communication between \(n\) workers, \(n\) tasks—for each message exchange. This fact is illustrated by the implementation of the BSP model as a dynamic task graph (§3.3.2).

In this section, I evaluate the performance of a variety of synthetic workloads, in order to establish the limits that the CIEL architecture places on scalability. Each of the synthetic workloads performs minimal computation, in order to isolate the effects that are caused by framework

\(^1\)This operating system is packaged for EC2 as Amazon Machine Image ami-2d4aa444.
code. All of the workloads in this section were implemented using the Scala implementation of distributed threads (§5.3.2), and evaluated on a 20-worker cluster of virtual machines as described above.

The workloads in this section are designed to answer the following questions:

- What is the minimum execution time for a task? What are the major components of this execution time? (§6.2.1)
- How is the minimum execution time for a task affected by load on the master? (§6.2.2)
- What is the minimum execution time for one iteration of an iterative algorithm? (§6.2.3)

### 6.2.1 Sequential task handling

In this experiment, I measured the end-to-end execution time for a single task. The synthetic workload is a single distributed thread, comprising 5000 tasks, as illustrated in Figure 6.1(a). Each task in the thread performs no computation and immediately blocks. Figure 6.1(b) shows the average task execution time using four versions of task blocking:

**Yield** The default implementation of thread yielding in Scala uses delimited continuations to construct a continuation object, and tail-spawns a task that depends on the continuation object (§5.3.2). The average execution time for a Yield task is 24 milliseconds. In this configuration, each Yield task is always scheduled on the same worker as its predecessor (using locality-aware scheduling, §4.3.1), and the JVM is reused between tasks.

**Suspend** This version uses the facility for legacy processes to block on a reference and remain pinned to a worker (§5.3.3). The average execution time for a Suspend task is 13 milliseconds. In this configuration, a delimited continuation is not constructed, and each Suspend task is pinned to the same worker as its predecessor. The JVM is reused between tasks, but it may only be used for the continuation of the task that previously suspended.

**Yield & Fetch** This version is the same as Yield, but it does not employ locality-aware scheduling, so the task may be scheduled on any worker (selected uniformly at random). The average execution time for a Yield & Fetch task is 36 milliseconds. In this configuration, the delimited continuation must (in most cases) be fetched from a remote worker before the task may begin. The JVM may be reused between tasks, although it is possible that the JVM may be garbage collected if no task is scheduled on a worker for a period of 30 seconds.

**Yield & VM** This version is the same as Yield, but it does not reuse the JVM between tasks, so the task execution must also include JVM startup time. The average execution time for a Yield & VM task is 805 milliseconds. In this configuration, each task is always
Figure 6.1: A synthetic sequential workload: a chain of tasks in which each task tail-spawns its successor.

scheduled on the same worker as its predecessor, and the delimited continuation must be read from the local object store before the task may begin.

The above versions of distributed thread blocking allow the costs and benefits of various CIEL features to be quantified. The use of delimited continuations incurs, on average, an overhead of 11 milliseconds, which can be attributed to the cost of invoking Java serialisation code, and writing to the local object store. Without locality-aware scheduling (or if the local worker is not idle), the remote object store read for the continuation object costs, on average, 12 milliseconds. For very small tasks, the benefit of JVM reuse is substantial: on average, over 780 milliseconds.
Furthermore, as discussed in §4.3.2, JVM reuse enables the use of in-memory caching, which is beneficial for workloads with large, frequently-used input data.

Figure 6.1(c) decomposes the task execution time into time spent in the worker and master. Since the task graph (Figure 6.1(a)) is serial, the time spent in the master can be calculated by measuring the time between a task result being received, and the next task being dispatched. Since the various versions of blocking have no impact on the processing in the master, the average time spent in the master across all versions is 2.0 milliseconds. The time spent in the worker follows the same pattern as overall execution time; however, note that the variance is much larger for end-to-end execution time than worker or master execution time, which suggests that the HTTP control traffic is responsible for the variance.\(^2\)

### 6.2.2 Parallel task handling

The previous experiment considered the “ideal” case in which only one task is runnable at once. In this experiment, I investigate the effect of varying the load on the master, due to spawning and dispatching tasks in parallel. Figure 6.2(a) shows the synthetic workload for this experiment: a single root task spawns \(N\text{\textsubscript{task}}\) distributed threads, which then each produce a chain of 5000 tasks, in the same manner as the thread in §6.2.1. Each thread uses the default Yield version of blocking, as described above.

\(^2\)Malte Schwarzkopf and I have recently begun to investigate ways of specialising the messaging protocol used in CIEL, in order to exploit non-cache-coherent multi-core computers [SMH11].
Figure 6.3: A synthetic iterative workload: multiple iterations of $N_{\text{task}}$ tasks followed by one aggregation task.

Figure 6.2(b) shows the effect of increasing the number of distributed threads from 1 to 20 (i.e. the cluster size). The time measured is the end-to-end task execution time, as perceived by each worker (i.e., since threads remain local to the same worker, this is the average time between a new task arriving at the worker). For $N_{\text{task}} = 1$, the task execution time is the same as the sequential version above—i.e. 24 milliseconds. However, as $N_{\text{task}}$ is increased to 20, the average task execution time increases to 211 milliseconds.

The increase in task execution time can be attributed to contention at the master: the current task graph and scheduler implementations are single-threaded and event-based, so parallel updates will be inserted into a queue and somewhat delayed. Note however that the throughput (in terms of tasks completed per second) increases to a maximum of 128 tasks per second when $N_{\text{task}} = 10$, which indicates that some I/O concurrency is being exploited. However, for $N_{\text{task}} > 10$, the throughput declines.

In this example, many parallel threads are making disjoint updates to the dynamic task graph. Therefore, it would be possible in future to increase the granularity of locking in the task graph, or use techniques based on lock-free data structures [FH07] or transactional memory [HM93], in order to improve throughput.

### 6.2.3 Iterative task handling

The third workload is based on a simple form of iterative computation. In this experiment, I investigate the limits to the performance of iterative algorithms, by constructing a synthetic
workload that has the same dependency structure as a simple parallel iterative algorithm. Figure 6.3(a) shows the dynamic task graph that is used in this experiment: each iteration comprises \( N_{\text{task}} \) parallel tasks, followed by a single task that depends on all of the parallel tasks. The single task is equivalent to a fixpoint task in the discussion in §5.2.3. As before, the tasks are implemented as Scala distributed threads, and the fixpoint task performs a default Yield between iterations.

In this experiment, the average iteration time is measured: this is measured as the average time between receiving results from fixpoint tasks at the master. Figure 6.3(b) shows the effect of increasing the number of parallel tasks on the average iteration time. With a single parallel task, the dynamic task graph degenerates into a chain, and the iteration time is slightly less than the end-to-end execution time for two tasks: 44 milliseconds. The execution time is less because one of the tasks does not spawn any child tasks, which simplifies handling at the master.

As in the previous experiment, increasing the number of parallel tasks increases the average iteration time: for \( N_{\text{task}} = 20 \), each iteration takes 375 milliseconds. The increase can be attributed to two factors:

- The fixpoint task must spawn a greater number of tasks, which means that its result message becomes larger, and requires more time to encode, transmit and decode.
- The task results from the parallel tasks will contend for the dynamic task graph data structure.

Since the fixpoint task spawns many almost-identical tasks (differing only in the data dependency), it is likely that the task result could benefit from compression. In order to improve CIEL performance when scaling to larger numbers of tasks, it may be desirable to support a “spawn template” or similar feature that makes it more efficient to spawn many similar tasks.

The minimum execution time for an iteration bounds the possible parallel speedup that a system can achieve, because it represents a serial overhead that cannot be ameliorated (and, indeed, worsens) as more workers are added to the system [Amd67]. However, sub-second performance for scheduling an iteration on 20 workers is acceptable for CIEL, especially when compared to similar systems. For example, in the following section, I will show that the minimum job length in Hadoop MapReduce is greater than 20 seconds, which means that CIEL achieves better efficiency than Hadoop for small jobs.

### 6.3 MapReduce

The next set of experiments investigate the performance of CIEL when running MapReduce-style jobs. Subsection 3.3.1 showed how any MapReduce job can be expressed as a dynamic task graph. The aim of this section is to investigate whether using a generic task graph to
implement a fixed dependency structure is more expensive than using a specialised framework. Two applications are considered: a CPU-intensive Monte Carlo estimation of π (§6.3.1) and an I/O-intensive distributed regular expression matcher (§6.3.2).

### 6.3.1 π estimation

π estimation is an example MapReduce application that is included with the Hadoop distribution. It is an example of a simple task-parallel algorithm in which every task contributes to computing a single aggregate value: in this case, an estimate for the value of π. In this algorithm, Monte Carlo sampling is used to estimate the probability that a point in a $2 \times 2$ square, chosen uniformly at random, falls inside a unit-radius circle that is inscribed on that square. π estimation is a useful benchmark, because it has two parameters that enable experimentation: the number of parallel tasks, $n_t$, and the number of samples, $n_s$. Overall, a job computes $N = n_t n_s$ samples. Holding $N$ constant and varying the parameters enables the parallel speedup to be discovered for a given problem size. Decreasing $n_s$ and holding $n_t$ constant increases the proportion of time per task spent in the framework, which enables the framework overhead to be evaluated.

To enable a fair comparison, the pi example application from Hadoop was ported to CIEL, using a Skywriting script that creates $n_t$ PiMapper tasks and one PiReducer task. The PiMapper and PiReducer tasks are Java-executor tasks that wrap the respective Hadoop Mapper and Reducer implementations.

The absolute performance of this algorithm is faster on CIEL than Hadoop for every configuration tested. Figure 6.4 shows the absolute performance of Hadoop and CIEL for different problem and cluster sizes. The execution time for the smallest problem size, $N = 100$, indicates the overhead that the frameworks impose on job execution: for 100 workers, this is approximately 28 seconds on Hadoop and 3 seconds on CIEL. In both cases, the implementation of the Monte Carlo step is identical, so the execution time for the largest problem size, $N = 10^{10}$, on a single worker indicates what overhead the framework imposes on the computation: this is approximately 3012 seconds on Hadoop and 1736 seconds on CIEL. The Hadoop overhead can be attributed to the implementation of the OutputCollector, which implements map-side reduction of the output samples; the CIEL version simply updates a local reduction variable.

CIEL also achieves better parallel speedup than Hadoop for the larger problem sizes\(^3\). Figure 6.5 shows the parallel speedup for the three largest problem sizes ($N \in \{10^8, 10^9, 10^{10}\}$). For

---

\(^3\)Interestingly, for small $N$, CIEL achieves a poorer speedup than Hadoop, because the Hadoop’s 27-second per-job overhead is relatively constant across cluster sizes, whereas CIEL’s execution time increases with additional tasks. The reason is that Hadoop uses a periodic heartbeat protocol to exchange tasks and results between the master and workers, and the period is fixed at 3 seconds for clusters that have fewer than 300 workers. On larger clusters, the period increases linearly as $N_w/100$, where $N_w$ is the number of workers.
Figure 6.4: Comparison of Hadoop and CIEL absolute performance for π estimation with $N \in \{10^2, 10^3, \ldots, 10^{10}\}$ and $n_t \in \{1, 10, 20, 50, 100\}$. Within each number of workers, each bar corresponds to a different value of $N$. Note that the $y$-axis has a logarithmic scale.

Figure 6.5: Comparison of Hadoop and CIEL parallel speedup for π estimation with $N \in \{10^8, 10^9, 10^{10}\}$. 

\( N = 10^8 \), CIEL achieves a \( 7.3 \times \) speedup on 20 workers; by contrast, the best Hadoop speedup is \( 2.26 \times \) on 50 workers. However, the corresponding execution times are 2.6 seconds on CIEL and 22.4 seconds on Hadoop, which are approximately the same as minimum job execution time on these platforms. For the largest job size, \( N = 10^{10} \), CIEL achieves an \( 80.7 \times \) speedup on 100 workers, compared to a \( 45.4 \times \) speedup on Hadoop.

### 6.3.2 Grep

The second MapReduce benchmark uses the Grep example application that is included with Hadoop to search a 22.1 GB dump of the English-language Wikipedia for a three-character string. The original Grep application performs two MapReduce jobs: the first job parses the input data and emits strings that match a given regular expression, and the second sorts the matching strings by frequency. As in the previous example, the mapper and reducer code was ported from Hadoop to CIEL, using a Skywriting script to compose the two jobs together. Both versions of the application use the same data formats and execute the same regular expression matching routine. In both versions, the input is divided into 165 chunks, to match the 128 MB default block size that Hadoop uses.

Figure 6.6(a) shows the absolute execution time for Grep as the number of workers increases from 1 to 100. CIEL achieves better performance than Hadoop for all cluster sizes. On a single worker, where all data is local, CIEL takes 36% less time than Hadoop: this saving arises because CIEL open local input files directly, whereas Hadoop accesses files indirectly via HDFS. The relative performance of CIEL improves as the number of workers increases: CIEL
takes 44% less time than Hadoop on 10 workers, and 76% less time on 100 workers.

Figure 6.6(b) shows the same results as parallel speedup ratios. CIEL also achieves better parallel speedup than Hadoop across all of the cluster sizes considered. Recall from §6.3.1 that a Hadoop job performing negligible computation on 100 workers runs for an average of 28 seconds. Since Grep involves two jobs, one would not expect Hadoop to complete the benchmark in less than 56 seconds. Given that the serial execution runs for $T_1 \approx 2580$ seconds, this therefore limits the maximum parallel speedup to $T_1 / 56 \approx 46$. The real speedup that Hadoop achieves is less than 46, because the 165 input chunks cannot be split equally between all workers. Indeed, the increased overhead of collecting results from 100 workers causes the speedup of Hadoop on 100 workers (18.4×) to be less than on 50 workers (24.7×). By contrast, CIEL achieves its best speedup on 100 workers—47.2× faster than serial execution, and a greater speedup than the theoretical maximum speedup that Hadoop can achieve.

The results in this section demonstrate that the expressivity of dynamic task graphs in CIEL does not impose a performance penalty on less-powerful, MapReduce-style computations, and the implementation achieves performance that is competitive with an industrial-standard MapReduce implementation. Furthermore, the results in this section reveal that CIEL is better suited to short jobs than Hadoop, which is a consequence of Hadoop’s original application, viz. large-scale document indexing. However, anecdotal evidence suggests that production Hadoop clusters mostly run jobs lasting less than 90 seconds [ZBSS+10], which suggests that they too may benefit from a more efficient system, such as CIEL.

### 6.4 Iterative $k$-means

The $k$-means clustering algorithm is an iterative algorithm for assigning $n$ data points $\in \mathbb{R}^d$ to $k$ clusters [Mac03, pp. 285–289]. Each cluster is represented by the centroid (or mean) of the points in that cluster. The algorithm starts with $k$ arbitrarily-chosen centroids; in each round, every data point is compared to the $k$ centroids and assigned to the cluster with the nearest centroid\(^4\). The algorithm iterates until the clusters converge to a fixed point: this may be detected when no points change cluster, or if the cluster centroids move by less than $\epsilon$ (for some $\epsilon > 0$) from one iteration to the next.

To compare the performance of CIEL and Hadoop for this algorithm, the Hadoop-based $k$-means implementation from the Apache Mahout scalable machine learning toolkit [Mah] was ported to run on CIEL. Whereas the Hadoop version uses a driver program to submit multiple jobs and perform the convergence test, the CIEL version uses a first-class Java task for the reducer, which computes the updated centroids, and only tail-spawns another iteration if the centroids move by more than a configurable value, $\epsilon$. Apart from this, the ported version uses the same data format and computational kernel as the Hadoop version.

\(^4\)A typical metric is squared Euclidean distance, because it may be computed using computationally-cheap floating-point instructions.
**Figure 6.7:** Comparison of Hadoop and CIEL MapReduce execution time for one $k$-means iteration. The average execution time per iteration is shown; error bars represent one standard deviation.

**Figure 6.8:** Comparison of MPI and optimised CIEL execution time of one $k$-means iteration. Note that the $y$-axis has a smaller scale than Figure 6.7, because the MPI version is approximately $40 \times$ faster than the MapReduce-based implementations.
In this experiment, the performance of the two versions is compared for different input sizes, running on 20 workers. The number of input vectors is increased from $n = 1.6$ million to $n = 32$ million. Each vector contains 100 double-precision values, and $k$ is set to 100 clusters. Figure 6.7 compares the per-iteration execution time for the two versions, in addition to a modified version of the CIEL job that uses in-memory caching. Each configuration was repeated five times, and the error bars show one standard deviation. For each job size, CIEL is faster than Hadoop, with the difference being approximately 128 seconds for the larger job sizes. The constant difference can be attributed to two factors: first, the constant overhead of Hadoop job creation, discussed in Section 6.3; and second, the reduced number of non-local tasks owing to the use of sweetheart references, discussed in Section 4.3.2. In-memory caching has little impact on the performance, saving at most 15% of the execution time for 8 million vectors. Furthermore, in this configuration it is not effective beyond $n = 24$ million vectors, because the total amount of data to be cached exceeds the available amount of physical RAM, which leads to thrashing—the enlarged error bars for $n = 24$ million indicate the early effects of thrashing, which would be magnified at larger input sizes.

Although MapReduce-based solutions are convenient, because of non-functional features such as fault tolerance and elastic scaling, they are unlikely to have the best absolute performance, because the framework and programming abstraction introduce various overheads. Therefore, the next experiment compares $k$-means performance to an optimised MPI implementation that uses direct message passing between processes that store the entire data in memory [Lia09]. Figure 6.8 shows the absolute performance of the MPI version. For $n = 1.6$ million vectors, the MPI version is $32.6 \times$ faster than the Hadoop-on-CIEL version and $57.8 \times$ faster than the original Hadoop version; and for $n = 32$ million, the MPI version is approximately $42 \times$ faster than both MapReduce-based versions.

To explain the large difference in performance between the MPI and MapReduce-based versions, note that the MPI code uses unchecked `double` arrays instead of vector objects, and an unchecked two-dimensional `double` array instead of a map-based data structure to store the partial sums for each partition. Such optimisations are not possible in a MapReduce programming model, because each vector is assigned to a cluster independently, and the framework is responsible for calculating the partial sums using a combiner function. The object-oriented design of the Java mapper and reducer tasks leads to a large number of virtual function calls per vector—which is difficult for the compiler to optimise—and yields poor performance on a modern superscalar processor, due to branch prediction misses [DH96]. This insight is the basis of my parallel work on Steno, which is a compiler for declarative queries that automatically eliminates much of this overhead by generating simple imperative loops [MIY11].

Since CIEL is not constrained to a MapReduce programming model, it is possible to implement the $k$-means algorithm using simple arrays to store the vectors and partial sums. The “CIEL optimised” series in Figure 6.8 shows the result of applying this optimisation to the $k$-means job. Note that this version still represents all of the state as CIEL objects, which are stored on disk. However, the performance is much improved, and is only $2 \times$ slower than the MPI version.
Figure 6.9: Distribution of time spent executing in the master for a chain of Yield tasks, with and without journalling enabled. The difference between the median observations is 0.26 milliseconds.

Moreover, the optimised version of $k$-means benefits proportionately more from the in-memory cache, because the ratio of I/O to computation is higher. The “CIEL optimised (cached)” series in Figure 6.8 exhibits a 30% improvement over the non-cached case, and iterations take only 40% more time than the MPI version.

These results lead to the conclusion that a distributed execution engine can achieve performance that approaches an optimised parallel implementation, while retaining the qualitative benefits of transparent fault tolerance, scheduling and elastic cluster membership. Furthermore, it may be possible to extract even better performance, by, for example, switching to a lower-level implementation language, such as C. In fairness to the Mahout implementation, its abstractions make it possible to use different vector representations and distance calculations, which would be more onerous to support in a lower-level language. However, these benefits should be traded off with the performance penalty of using an object-oriented coding style for CPU-intensive tasks.

6.5 Fault tolerance

Section 4.4 described the forms of fault tolerance that CIEL supports: client, worker and master fault tolerance. To recap, client fault tolerance allows an iterative job to continue without participation from the client; worker fault tolerance allows a job to continue if a subset of the workers fail; and master fault tolerance enables a replacement master to reconstruct the dynamic task graph in the event of a master failure.

In this section, I will investigate the costs and benefits of master fault tolerance. Master fault
tolerance incurs a cost because all spawned tasks and published references are logged either to a journal file or a hot standby. To evaluate the overhead and isolate the role of the master, I repeated the sequential Yield experiment from §6.2.1, with and without master fault tolerance (local journalling) enabled. Figure 6.9 shows the distribution of time spent in the master, for 800 tasks. The median without journalling is 1.8 milliseconds, and the median with journalling is 2.1 milliseconds. Therefore, the per-task overhead for master fault tolerance is approximately **0.26 milliseconds**, which is dominated by variations in network latency.

At a high level, the aim of master fault tolerance is to ensure that a computation continues to make progress in the presence of master failures. In CIEL, the failure model assumes that all data is either durably stored (e.g. replicated) or produced as a task output: this means that, in the worst-case scenario, it is possible to re-execute a job from the start. Therefore, the master fault tolerance scheme is only useful if it can complete a job, after the master resumes, in less time that it would take to restart the job from scratch.

In this experiment, I investigate the effects of master failure and fault tolerance on the end-to-end execution time of an iterative job. The job is based on the synthetic workload from §6.2.3, with the additional feature that each parallel task sleeps for three seconds to simulate work. For this experiment, \( N_{\text{task}} = 100 \), and the workload iterates 20 times. Therefore, on a cluster of 20 workers, one would expect the job to take at least 300 seconds: in fact, the average fault-free execution time is 332 seconds. After starting each job, a failure time, \( t_{\text{fail}} \in [0, 300] \) is chosen uniformly at random, and master failure is induced after \( t_{\text{fail}} \) seconds have elapsed. The master is then resumed, and the job restarted.

Figure 6.10(a) shows the relation between time spent executing before and after the master failure. The dotted diagonal line, \( x + y = 332 \), represents the ideal case, in which the total time spent executing is the same as a fault-free execution. The dashed horizontal line represents the worst case, in which the time spent executing after the master failure is always the same as restarting the job from scratch. The crosses represent test runs on the 20-worker cluster: from the figure it is clear that the actual performance is close to the ideal case. Figure 6.10(b) shows the overhead in more detail: each cross is now plotted as the percentage overhead in execution time, compared to the ideal case. Figure 6.10(b) shows that, for the 40 test runs, the overhead is less than 2.5% in every case.

Since task execution is atomic, the work lost due to failure becomes smaller if the task granularity is smaller: hence it is necessary to choose a sufficiently small task granularity when designing a computation to run on CIEL (or another distributed execution engine). In Subsection 6.6.2, I will demonstrate how choosing the appropriate task granularity is also vital for achieving good performance in fault-free execution.
Figure 6.10: The overhead of master fault tolerance under randomly-induced failures. In (a), the dotted diagonal line represents the ideal (no-overhead) case and the dashed horizontal line represents the pessimal (restart from scratch) case. (b) All of the observed test runs are within 2.5% of the ideal case.
6.6 Streaming

The role of streaming in CIEL (§4.2.3) is to provide a performance optimisation, by allowing a consumer to begin processing an object before it has been fully produced. In effect, streaming makes it possible to have finer-grained data dependencies than whole objects: once it is running, a task can block on a chunk of streaming data more efficiently than a whole object, because the system provides a direct peer-to-peer connection between the appropriate workers, and there is no need to send metadata to the master.

In this section, I will present two experiments that make use of the streaming feature. First, in Subsection 6.6.1, I will present a microbenchmark that evaluates the performance of two variants of streaming: direct TCP streams and indirect HTTP streams. Then, in Subsection 6.6.2, I will discuss how the Binomial option pricing model can use streaming to achieve fine-grained pipelined parallelism.

6.6.1 Microbenchmark

CIEL supports several different streaming modes, which have different performance and fault tolerance characteristics. In this experiment, I investigate the throughput of the direct TCP and indirect HTTP modes as the amount of data is increased. Figure 6.11(a) shows the dynamic task graph for the experiment: the task marked $t_P$ is the producer and the task marked $t_C$ is the consumer. The producer repeatedly writes up to 4 KB chunks of random data into the stream, and the consumer reads from the stream until it receives an end-of-file exception. Both the producer and consumer are implemented using the FCJava executor (§5.2.3).

Figure 6.11(b) shows the throughput achieved by the two modes for data sizes ranging from $10^3$ to $10^{10}$ bytes. For transfers less than $10^6$ bytes $\approx 1$ MB, the throughput is negligible, because the transfer time is dominated by the time taken to set up the channel. For larger transfers, the two modes are equal until $10^7$ bytes, at which point the direct TCP connection becomes much faster. For $10^{10}$ bytes $\approx 10$ GB, the direct TCP connection achieves 89% of the throughput reported by iperf; the overhead is due to an additional user-space copy through CIEL at both ends of the pipe\(^5\).

The performance of the indirect HTTP stream degrades after $10^9$ bytes: because the indirect mode writes all produced data to the file system, this mode is able to exploit the Linux buffer cache when the object fits in memory. However, because the virtual machines in this evaluation have only 1.7 GB of RAM (§6.1), the largest object will not fit in the buffer cache, and so performance will be degraded by the need for disk I/O.

Although the performance of indirect HTTP is poorer than a direct TCP connection, the indirect HTTP mode has some qualitative advantages. For example, in some cases, the indirect HTTP

---

\(^5\)Using the most recent version of the CIEL executor interface, it is possible for a task to receive a file descriptor to an open socket, but this feature is not currently supported in Java.
Figure 6.11: Streaming throughput in direct TCP and indirect HTTP modes. The graph shows the median throughput of 8 runs. The dotted horizontal line represents maximum TCP throughput, as measured by iperf.

mode will allow the producer to finish earlier, because there is no backpressure on the producer’s output. Furthermore, because the indirect HTTP mode writes a full copy of the object to disk, the data will remain in the event of a consumer needing to re-execute. Finally, it is easier to have multiple consumers in indirect HTTP mode, because the producer can start producing immediately without having to wait for a potentially-large number of consumers.

Note finally that, although the streaming facility is a useful performance optimisation for some algorithms, such as the one below, the dynamic task graph model prevents it from dealing with infinite streams. I discuss some extensions to the model that would permit such computations in Chapter 7.
6.6.2 Binomial option pricing

The binomial options pricing model (BOPM) is a dynamic programming algorithm for computing the expected value of a stock at a future date, based on assumptions about stock market trends and the risk-free interest rate [CRR79]. The algorithm computes every element in a binomial tree, which can be represented as an upper-triangular matrix, $P$. The rightmost column of $P$ can be computed directly from the input parameters, after which element $p_{i,j}$ depends on $p_{i,j+1}$ and $p_{i+1,j+1}$, and the overall result is $p_{1,1}$. Parallelism is achieved by dividing the matrix into row chunks, creating one task per chunk, and streaming the top row of each chunk into the next task. Figure 6.12 shows the element- and chunk-level data dependencies for this algorithm.

BOPM is not an embarrassingly parallel algorithm. However, one would expect CIEL to achieve some speedup, since rows of the matrix can be computed in parallel, and streaming tasks can be used to obtain pipelined parallelism. It is also possible to achieve better speedup by increasing the resolution of the calculation: the problem size ($n$) is inversely proportional to the time step ($\Delta t$), and the serial execution time increases as $O(n^2)$.

Figure 6.13 shows the parallel speedup of BOPM on a 50-worker CIEL cluster. The number of tasks is varied, and $n$ is increased from $2 \times 10^5$ to $1.6 \times 10^6$. As expected, the maximum speedup increases as the problem size grows, because the amount of independent work in each task grows. For $n = 2 \times 10^5$ the maximum speedup observed is $4.7 \times (30$ tasks), whereas for $n = 1.6 \times 10^6$ the maximum speedup observed is $26.0 \times (180$ tasks). After reaching the maximum, the speedup decreases as more tasks are added, because smaller tasks suffer proportionately more from constant per-task overhead. Due to the implementation of streaming ($\S$4.2.3), the minimum execution time for a stream consumer is approximately one second; therefore, in the limit, the job execution time will be $N_t$ seconds, where $N_t$ is the number of tasks.
Figure 6.13: The maximum parallel speedup for BOPM increases as the problem size is increases, because the overall amount of work grows quadratically. The speedup initially increases when the number of tasks, before plateauing and eventually declining as the task overhead begins to dominate.

6.7 Summary

In this chapter, I have investigated the performance of CIEL when executing a mixture of synthetic and realistic jobs. The main conclusions are as follows:

- The performance cost of data-dependent control flow is negligible. On 20 workers, the average end-to-end execution time of one iteration in a minimal iterative computation is less than 400 milliseconds.

- The expressive power of dynamic task graphs does not impose a significant overhead on static computations. For MapReduce-style jobs, CIEL achieves better performance than Hadoop on both data-intensive and CPU-intensive jobs.

- The ability to use different programming models in CIEL enables optimisations that, for some jobs, approach the performance of MPI-style code. In particular, eliminating abstractions that make heavy use of virtual calls (such as MapReduce) leads to a large improvement in performance.

- The streaming (§4.2.3) and master fault tolerance (§4.4.3) features of CIEL enable an efficient use of cluster resources.

Although these results are encouraging, one must be cautious when interpreting the comparative results. Both CIEL and Hadoop are large systems, the implementations of which contain many design decisions that may affect performance while being orthogonal to the programming
model. A good example is Hadoop’s piggybacking of task status messages on the periodic heartbeat messages, which protracts the duration of small tasks. It would be possible to implement a version of MapReduce that uses a more efficient control protocol, and would achieve performance that is closer to, or better than CIEL. Likewise, the current implementation of CIEL uses an HTTP-based RPC protocol that uses a text-based encoding of control messages, but it is possible to use other encodings and protocols (such as dedicated message-passing hardware [SMH11]).

Nevertheless, the current implementation of CIEL achieves good performance on the coarse-grained data- and CPU-intensive problems for which it was developed, and the programming model and fault tolerance represent an advance over existing systems. In the next and final chapter, I will discuss extensions to CIEL and the dynamic task graph model that will enable them to tackle an even greater range of problems.
Chapter 7

Conclusions and future work

With the growth of data-centre computing, shared-nothing compute clusters have become commonplace and—through cloud computing providers—accessible even to individuals with modest resources. Therefore, motivated by the widespread adoption of the MapReduce and Dryad programming models, I have sought in this dissertation to expand the class of algorithms that can easily be executed on these clusters. In order to achieve this aim, I have made the following contributions:

- In Chapter 3, I introduced the dynamic task graph execution model, which subsumes static models (such as MapReduce and Dryad) and extends the notion of spawning new tasks as a first-class operation available to all tasks in the computation. This enables a dynamic task graph to perform data-dependent control flow, which allows it to represent iterative or recursive algorithms. I further demonstrated the Turing-completeness of the model by showing a reduction from while programs to dynamic task graphs.

- In Chapter 4, I presented CIEL, which is a distributed implementation of the dynamic task graph model for data-intensive computations. While Chapter 3 discussed dynamic task graphs from a functional perspective, Chapter 4 focused on non-functional aspects, such as scheduling, I/O performance and fault tolerance. CIEL includes a scheduler that attempts to reduce the amount of data transferred over the network. CIEL also provides greater fault tolerance than MapReduce, Dryad or any similar execution engine, because it tolerates master and client failures in addition to worker failures. In Chapter 6, I showed that CIEL implements data-dependent control flow with negligible impact on performance, and the system achieves performance that is competitive with systems that expose a less-powerful execution model.

- Having demonstrated that CIEL can execute dynamic task graphs efficiently, in Chapter 5 I showed how the distributed thread programming model can be built atop the system. The key idea in this chapter was to implement synchronisation (blocking) by storing the current continuation as an object, and spawning a task that depends on both the continuation
and the synchronisation target. I developed these ideas for several platforms, including a new scripting language called Skywriting, the Scala programming language, and arbitrary processes using process checkpointing. In addition to distributed threads, CIEL supports simpler programming models that support static computations, and the implementation allows developers to mix and match these approaches in the same job.

Collectively, these contributions constructively prove the thesis that I stated in Chapter 1: it is possible to implement data-dependent control flow in a distributed execution engine, and the resulting system has the same—indeed a greater set of—desirable non-functional properties than existing, less-powerful execution engines. The dynamic task graph model is Turing-complete: it can express any effectively-calculable function, and it can also express arbitrarily-large degrees of parallelism.

At this point, it is tempting to conclude that dynamic task graphs are not only universal, but the ultimate execution model for distributed data-flow computing. However, although there is no algorithm that cannot be expressed as a dynamic task graph, as Peter Wegner noted, algorithms are not the sole form of computation [Weg97]. Furthermore, the present implementation of CIEL is designed to perform coarse-grained data-parallel computations in a commodity cluster, but other parallel architectures—such as multi- and many-core servers—are becoming commonplace. In the remainder of this chapter, I discuss possible extensions to CIEL, its execution model and its programming models.

### 7.1 Extending dynamic task graphs

To test the limitations of dynamic task graphs, one must revisit the definitions in Chapter 3. I defined tasks—or, more precisely, the abstract executor that is applied to a task—to be deterministic. I also defined objects—and hence the inputs to and outputs from each task—to be finite. These limitations have practical justifications that are found also in MapReduce [DG04] and Dryad [IBY07]: deterministic task execution ensures that a task can be re-executed in case one of its outputs is lost due to failure; finite-length objects ensure that each task input can be stored in case it is needed for re-execution of that task.

However, this limits the ability of dynamic task graphs to represent many realistic applications. For example, a dynamic task graph cannot efficiently represent a stream processing application, in which computation is performed on an infinite stream of inputs, because it is impractical and inefficient to store the entire contents of a stream in the CIEL object store. Soulé et al. have recently developed a universal calculus for stream processing languages [SHG+10], and it seems likely that there would be fruitful research in extending the dynamic task graph model to support these languages. For example, Soulé et al. assume the existence of stateful tasks and feedback cycles, so a possible approach would be to investigate means of supporting cycles in a dynamic task graph.
The requirement for determinism stems from a desire to support completely transparent fault tolerance. However, many algorithms can be implemented more efficiently if some amount of non-determinism is permitted. For example, the class of asynchronous algorithms allows local partitions to make progress without a global barrier, with global coordination performed asynchronously [CM69]. Furthermore, interactive applications are inherently non-deterministic, because their behaviour depends on an infinite stream of input events. It would be desirable to support such applications on a distributed execution engine like CIEL, because the system can still take care of many mundane details of setting up a distributed computation. However, it would be necessary to relax the requirement for deterministic task execution, and abandon the goal of completely transparent fault tolerance. I have carried out some preliminary investigations into this topic [MH11], and one promising approach involves bounded non-determinism, whereby subgraphs of a dynamic task graph have internal non-determinism but still have deterministic outputs.

7.2 Alternative system architectures

CIEL was developed as a successor to systems like MapReduce and Dryad, which in turn were developed to run on commodity clusters. This choice has influenced several further design decisions: for example, tasks synchronise by sending messages to the master, and—as discussed above—task execution must be deterministic in order to handle failures.

Ranger et al. evaluated the use of MapReduce for shared-memory parallel programming, and concluded that—when using a specialised version of the MapReduce runtime—the programming model was useful for implementing shared-memory computations [RRP+07]. Since dynamic task graphs are more expressive than MapReduce, it would be instructive to consider how CIEL could be changed to function more efficiently in a shared-memory environment. Malte Schwarzkopf and I have carried out some preliminary investigations in this area [SMH11], and we showed that, for an iterative microbenchmark (similar to the one used in §6.2.3), the per-iteration overhead can be reduced by exploiting sharing between cores in a 48-core server. Our long-term goal is to develop CIEL into a multi-scale execution engine, where the system coordinates parallel execution at both the intra- and inter-machine level. Given that execution within a single machine is (currently) assumed to be reliable, there would be useful synergy with the non-deterministic extensions that I described above.

It would also be possible to extend CIEL to much larger clusters that are distributed between multiple data centres. In a large online service, customer data is often geographically distributed, in order to reduce the latency for users and avoid correlated failure in a single data centre [CRS+08]. To extend CIEL to such environments, it may be necessary to develop a distributed master comprising nodes in several data centres. This could be achieved by partitioning the object and task tables in the existing master, but it will probably be necessary to develop new scheduling algorithms to mitigate the cost of transferring data between data centres.
Figure 7.1: Possible representation for non-deterministic select in an extended dynamic task graph. The task will become runnable when all of \( \{d_1, d_2, d_3\} \) become concrete, and then any of \( \{n_1, n_2, n_3\} \) becomes concrete or the timer expires.

### 7.3 Separating policy from mechanism

As in an operating system [LCC+75], there is no single set of policies that is optimal for all computations in a distributed execution engine. In this dissertation, I have primarily discussed and evaluated mechanisms; the sole exception is the scheduling policy (§4.3), which is designed to prefer computation on local data over transferring data across the network. Nevertheless, production-quality execution engines [DG04, IBY+07, Had] must choose among several policies for issues such as straggler detection [ZKJ+08, AKG+10], fair sharing [IPC+09, ZBSS+10] and data placement [AAK+11]. Taking the example of straggler detection, a policy that effectively detects stragglers in a homogeneous workload may lead to poor performance when the workload has inherently skewed task durations. Therefore, it would be desirable to allow developers to choose and implement application-specific policies, while providing a sensible default.

The distributed thread programming model is attractive for implementing these policies, because it allows the programmer to write arbitrary straight-line code, which then runs in the cluster. However, the current model is too conservative: for example, it is impossible to implement straggler detection in a distributed thread, because all synchronisation is deterministic, and hence it is impossible to observe the outcome of a “race” between tasks. One possibility would be to augment dynamic task graphs with a non-deterministic select dependency (Figure 7.1), which would cause a task to become runnable when any (as opposed to all) of its dependencies become concrete. The programming model could then resemble the select() system call [The08]. Clearly, this would require some support for non-determinism, as discussed in Section 7.1.

Taking this idea further, it may be desirable to customise the implementation of lower-level features in the system. For example, in CIEL a task becomes runnable when all of its dependencies are stream references, but the task may not be able to perform useful computation until it receives the last byte of each stream, so it may block for a long (though finite) period of time while consuming a worker. For such a task, it would be useful to redefine runnability to hold
only when all of the dependencies are fully-concrete references. In the current architecture, this dependency resolution code is in the heart of the master, where it would be undesirable to run arbitrary user-provided code (assuming that the cluster has many mutually distrusting users). To allow custom policies safely at this level, it would be necessary to *disaggregate* the master so that each job is coordinated in a separate address space, and the remaining master would resemble a microkernel that only performs request routing and task dispatch.

### 7.4 Summary

The first general-purpose computers were developed to perform repetitive calculations more efficiently. At that time the full range of applications, which we now take for granted, could not have been imagined. In this dissertation, I have presented a general-purpose distributed execution engine that performs repetitive calculations even more efficiently. I do not presume to know the full range of applications that CIEL or its successors will one day support.
Bibliography


ACM SIGPLAN conference on the History of Programming Languages (HOPL), 2007. See p. 29.


