UiT
THE ARCTIC UNIVERSITY OF NORWAY

Faculty of Science and Technology
Department of Science and Technology

COMBUSTI/O

Abstractions facilitating parallel execution of programs implementing common I/O patterns in a pipelined fashion as workflows in Spark

—

Jarl Fagerli

INF-3981 Master's Thesis in Computer Science – June 2016

Abstract

In light of recent years' exploding data generation in life sciences, increasing downstream analysis capabilities is paramount to address the asymmetry of innovation in data creation contra processing capacities. Many contemporaneously used tools are sequential programs, ofttimes including convoluted dependencies leading to workflows crashing due to misconfiguration, detrimental to both development efforts and production, also inducing duplicate work upon re-execution.

This thesis proposes a distributed and easy-to-use general framework for workflow creation and ad hoc parallelization of existing serial programs. In furtherance of reducing wall-clock time consumed by big data processing pipelines, its processing is horizontally scaled out, whilst supporting recovery and tool validation. COMBUSTI/O is a cloud and HPC ready framework for pipelined execution of unmodified third-party program binaries on Spark. It supports tool requirements of named input and output files, usage and redirection of standard streams, and combinations of these, as well as both coarse and fine granularity state recovery. Designed to run independently, its scalability is reduced to Spark and the underlying fault-tolerant big data frameworks.

We evaluate COMBUSTI/O on real and synthetic workflows, demonstrating its propriety for facilitation of complex compute-intensive workflows, as well as its applicability for data-intensive and latency-sensitive workflows, and validate the coarse-grained recovery mechanism and its cost for the different flavors of workflows. We show stage recovery to be beneficial during development, for compute-intensive workflows, and for error-prone data-intensive workflows. Moreover, we show that the I/O overhead of COMBUSTI/O grows for data-intensive workflows, and that our remote tool execution is inexpensive.

COMBUSTI/O is open-sourced at `https://github.com/jarlebass/combustio`, and currently used by SfB at the University of Tromsø.

To Helge and Anette

Acknowledgements

First and foremost I would like to express my sincerest gratitude to my advisor, Associate Professor Lars Ailo Bongo, for his guidance and assitance for the entire duration of this work. Appreciation is also extended my co-advisors, Inge Alexander Raknes and Giacomo Tartari, for their valuable technical- and design-related help and support.

I wish also to thank Edvard Pedersen, Jon Ivar Kristiansen and Espen Mikal Robertsen for their assistance with the cluster and parametrization of bioinformatics tools throughout the course of this work, and Dr. Erik Hjerde for sharing his insights and expertise in the field of biology. Moreover, I would like to express my gratitude to Jan Fuglesteg for helping me solve any and all practical and administrative issues encountered over the years.

Thanks to my flatmate Marius Larsson, the guys at the office, Bjørn Fjukstad, Einar Holsbø and Morten Grønnesby, and my fellow students, particularly Ruben Mæland and Eirik Mortensen. These years at the University has been great because of you!

To my father Helge, and my sister Tonje: Thank you for always believing in me and for the invaluable emotional support; I would not have gotten far without you.

Ultimately, I would like to express my heartfelt gratitude to Anine Elida Walbeck for her unwavering patience, encouragement, and loving support.

Contents

Abstract i

Acknowledgements v

List of Figures xi

List of Tables xiii

List of Code Listings xv

List of Abbreviations xvii

1 Introduction 1
- 1.1 Problem Context . . . 3
- 1.2 Challenges . . . 4
- 1.3 Requirements . . . 5
- 1.4 Proposed Solution . . . 7
- 1.5 Contributions . . . 8
- 1.6 Outline . . . 10

2 Architecture 11
- 2.1 The Architecture Stack of COMBUSTI/O . . . 12
- 2.2 Internal Components of COMBUSTI/O . . . 14
- 2.3 The New META-pipe Architecture . . . 15

3 Design and Implementation 19
- 3.1 Internal Design of COMBUSTI/O . . . 19
  - 3.1.1 Evaluation Workflow . . . 21
  - 3.1.2 Tool Abstraction . . . 24
  - 3.1.3 Tool Wrapper . . . 25
  - 3.1.4 Workflow Manager . . . 26
- 3.2 Framework Stack of COMBUSTI/O . . . 28

4 Use Case: Marine Metagenomics 31

4.1 Use Case and META-pipe 1.0 31
4.2 Implemented Workflow and Tools Wrapped 32
4.2.1 Ray Wrapper 34
4.2.2 MGA Wrapper 35
4.2.3 BLAST Wrapper 35
4.2.4 InterProScan 5 Wrapper 37
4.2.5 Marine Metagenomics Workflow 37
4.3 Implementation 39

5 Evaluation 41
5.1 Methodology 41
5.1.1 Cluster Specs and Configuration 42
5.1.2 Spark Configuration 43
5.1.3 Compute-Intensive Workflow 46
5.1.4 Data-Intensive and Latency-Sensitive Workflows . . . 49
5.1.5 Measurements 53
5.2 Results and Discussion 53
5.2.1 Compute-Intensive Pipeline Evaluation 54
5.2.2 Data-Intensive Pipeline Evaluation 56
5.2.3 Latency-Sensitive Pipeline Evaluation 60
5.2.4 Spark and I/O Tuning 63

6 Related Work 65
6.1 Frameworks Utilized 65
6.1.1 Apache Hadoop 65
6.1.2 Apache Spark 66
6.1.3 Network File System 68
6.2 Bioinformatics Pipelines and Frameworks 69
6.2.1 META-pipe 1.0 69
6.2.2 ADAM 71
6.2.3 CloudBurst 72
6.2.4 Crossbow 72
6.3 Biology and Bioinformatics Glossary 73
6.4 Bioinformatics Tools Wrapped 75
6.4.1 Ray 75
6.4.2 MetaGeneAnnotator 76
6.4.3 Basic Local Alignment Search Tool 76
6.4.4 InterProScan 5 77
6.5 *Tara* Oceans 78
6.6 ELIXIR 78

7 Conclusion 81

8 Future Work 85

Appendices 89

A Source Code 89

Bibliography 91

List of Figures

1.1 EMBL-EBI data growth by platform 2
1.2 DNA sequencing cost over the past years 3

2.1 Architectural components of COMBUSTI/O 12
2.2 Internal components of COMBUSTI/O 14
2.3 Architecture of the new META-pipe 16

3.1 Top-down view of COMBUSTI/O's design 20
3.2 Example workflow stages 21
3.3 Directory structure created per tool and stage 25
3.4 Scheduling, dissemination, and execution using Spark 27
3.5 The framework stack utilized in COMBUSTI/O 29

4.1 Stages of the use case workflow 33
4.2 Pipelined execution of the stages 38

5.1 Basic Spark configuration architecture 45
5.2 Results of the different compute-intensive configurations . . 55
5.3 Data-intensive results of the workflows 57
5.4 Latency-sensitive plot of the different workflows 61
5.5 Stacked plot of disk I/O of the binary execution workflow . . 63
5.6 CPU utilization of the binary execution workflow 64

6.1 Schematic overview of META-pipe 1.0 70

List of Tables

4.1 Ray arguments 34
4.2 `mpirun` arguments 35
4.3 `blastx` arguments 36
4.4 InterProScan 5 arguments 37
4.5 Lines of code per module 39

5.1 Hardware specifications of a type *A* `ice2` node 42
5.2 Hardware specifications of a type *B* `ice2` node 42
5.3 Accumulated cluster metrics 43
5.4 Data-intensive results of the workflows 58

6.1 Examples of transformations in Spark 67
6.2 Examples of actions in Spark 68

List of Code Listings

3.1 UNIX shell evaluation pipeline 21
3.2 Pragmatic Spark evaluation pipeline implementation 22
3.3 Accurate UNIX shell evaluation pipeline 22
3.4 Example workflow implementation 23
3.5 Execution logic interface 24
3.6 Tool wrapper implementor execution logic 26
3.7 Example `map` for distributing and disseminating tool execution 28

5.1 `spark-submit` options used to deploy 28 containers on `ice2` . 44
5.2 Full Ray command 47
5.3 Full MGA command 47
5.4 Full `blastx` command 48
5.5 BLAST database creation command 48
5.6 Full InterProScan 5 command 48
5.7 COMBUSTI/O Scala built-in workflow implementation . . . 52

6.1 Example *k*-mers 75

List of Abbreviations

AHCI Advanced Host Controller Interface

API application programming interface

ASCII American Standard Code for Information Interchange

AWS Amazon Web Services

BDAS Berkeley Data Analysis Stack

BDPS Biological Data Processing Systems

BLAST Basic Local Alignment Search Tool

BWT Burrows-Wheeler transform

CLI command-line interface

CPU central processing unit

CWL Common Workflow Language

DAG directed acyclic graph

DFS distributed file system

DNA deoxyribonucleic acid

DRAM dynamic random-access memory

DRY don’t repeat yourself

EBI European Bioinformatics Institute

EMBL European Molecular Biology Laboratory

ENA European Nucleotide Archive

FTP File Transfer Protocol

GC garbage collection

GUI graphical user interface

HDD hard disk drive

HDFS Hadoop Distributed File System

HPC high-performance computing

HTTP Hypertext Transfer Protocol

I/O input/output

IaaS infrastructure as a service

IPC inter-process communication

IUPAC International Union of Pure and Applied Chemistry

JAXB Java Architecture for XML Binding

JIT just-in-time

JMH Java Microbenchmark Harness

JMS Java Message Service

JVM Java Virtual Machine

LRU least recently used

MBRG Molecular Biosystems Research Group

MCA modular component architecture

METAREP Metagenomics Reports

MGA MetaGeneAnnotator

MPI Message Passing Interface

MSP maximal segment pair

NCBI National Center for Biotechnology Information

NeLS Norwegian e-Infrastructure for Life Sciences

NFS Network File System

NGS next-generation sequencing

NHGRI National Human Genome Research Institute

NIST National Institute of Standards and Technology

NVM Non-Volatile Memory

OS operating system

PaaS platform as a service

PCI Peripheral Component Interconnect

RAID redundant array of independent disks

RBS ribosomal binding site

RCN Research Council of Norway

RDD resilient distributed dataset

REPL read–eval–print loop

REST representational state transfer

RNA ribonucleic acid

RPC remote procedure call

S3 Simple Storage Service

SaaS software as a service

SfB Center for Bioinformatics

SNP single-nucleotide polymorphism

SQL Structured Query Language

SSD solid-state drive

UI user interface

VFS Virtual File System

VM virtual machine

WARC Web archive

XDR External Data Representation

XML Extensible Markup Language

YARN Yet Another Resource Negotiator

Introduction

The advent of low-cost, high-throughput deoxyribonucleic acid (DNA) sequencing was by and large the impetus establishing biology as one of the big data sciences. With affordable, widely available, and high throughput technology [1, 2] – contrary to its predecessor, the "first-generation" automated Sanger method [3] – the inception of next-generation sequencing (NGS) allowed an increasing number of laboratories to acquire the equipment necessary for producing voluminous quantities of raw sequencing data, resulting in a substantial increase in data growth rate. Historically, over the past decade, nearly a doubling of DNA sequencing data has been observed every seven months, and the sequencing capacities are projected to continue rapidly increasing over the next ten years [4]. The sheer vastness of unexplored data in life sciences and omics holds promise of great amounts of novel biological, medicinal, and evolutionary knowledge [1, 5, 6], but researchers are now confronted with several challenges in regard to handling and processing this distributed and heterogeneous data.

In order to explore and extract information of biological value from raw DNA sequencing data, it must undergo extensive refinement. This refinement usually consists of deep computational pipelines requiring substantial compute and storage resources for the different processing stages, involving various bioinformatics software [7], necessitating aptitude in both biology and computer science to operate. Thence, biologists are often times impeded by inadequate knowledge and experience with big data systems and frameworks, and if proficient in handling big data, the compute resources needed might not be readily

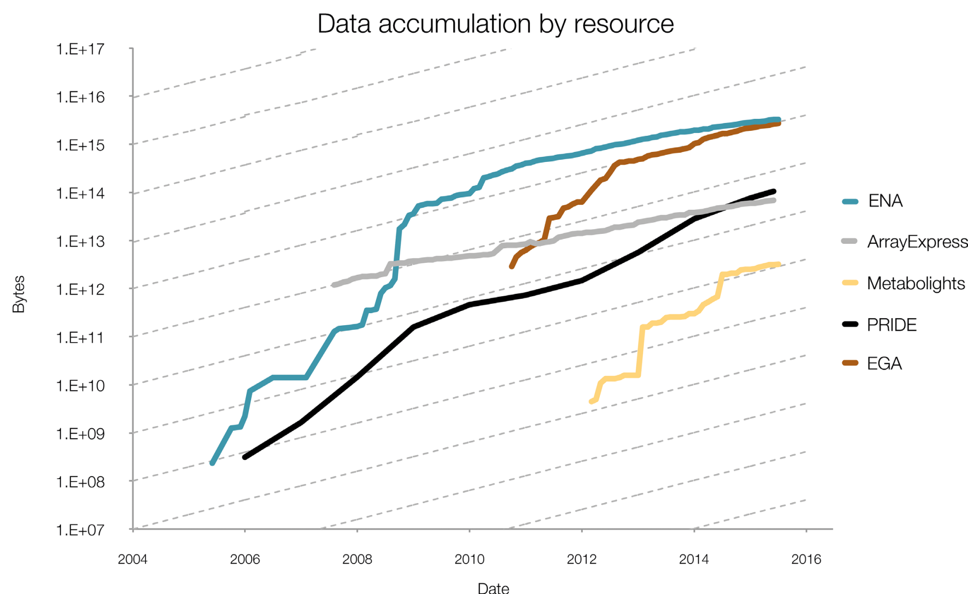

Figure 1.1: EMBL-EBI data growth by platform. Derived from Cook et al. [9]

available. Barring the aforementioned issues, non-trivial engineering matters still remain to be addressed, e.g., considerable knowledge of hardware and resource consumption to estimate infrastructure needs.

To contextualize scale, one of the largest genomic data repositories in the world, sustained by the European Molecular Biology Laboratory (EMBL)-European Bioinformatics Institute (EBI) [8], stored an approximate 75 petabytes of biological data and back-ups as of December 2015 [9]. Figure 1.1 shows the contents of the EMBL-EBI data platforms and their growth trends over the past decade. Correlative to the increase in data generation, the DNA sequencing cost dramatically decreased in roughly the same period, as shown in Figure 1.2.

Not having resources readily at hand may be amended by the increasing popularity of cloud services, and the current big data analysis trend has shifted to favor cloud-based solutions [5, 12]. Cloud computing vendors grants anyone access to compute and storage resources without needing to buy and maintain hardware by supporting ad hoc creation and configuration of clusters in which only the resources used are paid for, using infrastructure as a service (IaaS) providers like Amazon Web Services (AWS) [13], Microsoft Azure [14], and cPouta [15]. An important advantage of operating in the cloud is unifying compute and storage resources by storing all data in the cloud, effectively moving the data closer to the compute resources. This centralization allows for analyses on data from a multitude of sources, promotes sharing, and circumvents the need to pull data from remote sources at limited network bandwidths. More-

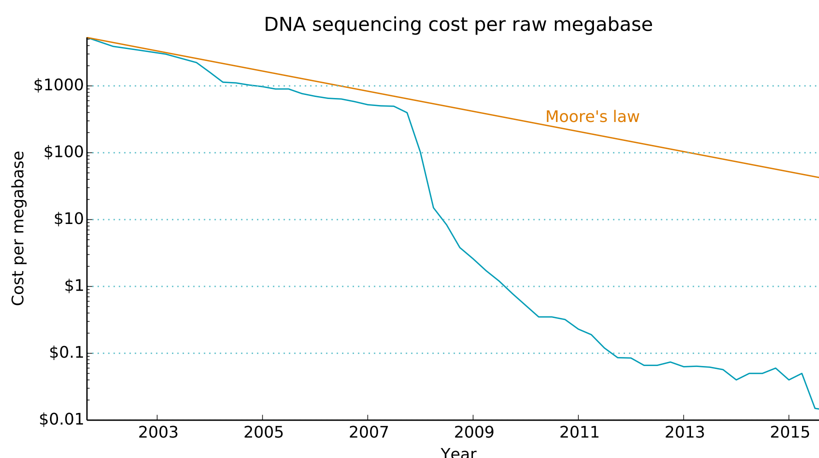

Figure 1.2: DNA sequencing cost over the past years. Based on data from the National Human Genome Research Institute (NHGRI) [10], inspired by [11]

over, most cloud service providers support elastic provisioning of resources on demand, which enables the adjustment of resources available to a service at a given time based on some predicate, e.g., scaling out or down in response to workload intensity. The cloud also serves as a neat abstraction for scientists unfamiliar with computer science, and having services deployed in the cloud with nice and intuitive graphical user interfaces (GUIs) may help eradicate concerns researchers of less technical prowess might have about using new systems, thus catering to a broader audience.

In consideration of the heretofore discussed, efficient and easy-to-use big biological data management and analysis approaches for cloud and HPC platforms are needed to alleviate portions of the bioinformatics bottlenecks and assist in the field's advance, and the work of this thesis addresses some of the challenges pertaining to these aspects. Specifically, we have implemented a backwards-compatible framework facilitating parallel execution of legacy program binaries implementing common I/O patterns in a pipelined fashion on top of widely used and established big data frameworks capable of performing metagenomic data analysis at scale.

1.1 Problem Context

This project is done in the context of the ELIXIR [16] European infrastructure for life sciences, wherein the University of Tromsø is leading a scientific use case in marine metagenomics. This use case is one of four demonstrations of the

technical services developed in ELIXIR, and is intended to help drive innovation and research within the field. It involves defining gold standard tools to be used in a domain-specific data analysis pipeline, as well as establishing a marine reference database, which are both to be deployed as software as a service (SaaS) (if categorized according to the cloud nomenclature of the National Institute of Standards and Technology (NIST) service models [17]), forming the most essential contributions of the University of Tromsø team.

Concomitant with actualizing the marine reference database MarRef is the gold standard data analysis pipeline META-pipe, which is an automated pipeline for annotation and analysis of metagenomic and genomic sequence data used to generate the results populating the reference database. However, some frailties were identified in the currently implemented version of the pipeline, among the most salient were the burden of manual failure handling, having built a distributed workflow manager from scratch: The runtime system for data management and parallel job execution is implemented using scripts, currently exceeding 10.000 lines of Perl code, and is becoming increasingly error-prone.

Current research efforts are therefore directed towards developing an improved version of META-pipe, revising its current flaws, chiefly addressing failure handling and provenance management, but also taking user interface (UI) and portability concerns into account. Efforts to remedy these issues prompted a revamp of the architectural components and structure of META-pipe 1.0, forming the basis of the new version currently in development, in which the work conducted in this thesis is an important part with regard to the backend processing component.

1.2 Challenges

Based on our experience developing and operating META-pipe 1.0, we have identified several issues and challenges for large-scale biological data analyses on distributed platforms. Program development for distributed environments and systems, i.e., adding new dependencies, functionality, updates, and bug fixes, can be strenuous with regards to testing and troubleshooting, as well as being time consuming. Distributed pipelines introduces another issue in which missing dependencies or bugs may be left undiscovered until the stage at which they are required or stumbled upon, possibly occurring after hours of computation in former stages. This behavior is detrimental not only to productivity, as valuable time is wasted, but is also a nuisance to the developers with regards to testing and debugging. Thence, we postulate that mechanisms for validation and recovery to abate these time sinks would prove advantageous germane to easing the testing and further development of such analysis pipelines.

Determining what tools to employ in an analysis pipeline for the marine domain is done based on two principal criteria and the trade-off between the two in combination. The first criterion is analysis result quality, as the selected tools should produce biological results of the highest quality for marine metagenomics; the second criterion is scalability, considering the tools chosen must scale to the size of the largest marine metagenomic datasets (§ 6.5) and preferably beyond, recognizing the anticipated data growth as previously emphasized. The latter is commonly neglected, as tool developers tend to accentuate analysis result quality over optimizing for large-scale performance. In addition, several tools are the outcomes of research projects and ofttimes have not been through proper software quality control and hence are not production ready. For instance, various bioinformatics tools disobey norms by returning inaccurate exit statuses, complicating their execution and error-checking. Furthermore, reimplementing and maintaining locally optimized tools is impractical, thus the de facto standard, unmodified, and regularly maintained biological tools are given a priority bias upon consideration.

Optimally, the tools should be selected based on the two aforementioned criteria in isolation oblivious to implementational details, relying on the underlying processing component to handle the complexities of integration without imposing considerable nonproductive overhead and system-imposed delays. Accordingly, one of our design goals is to be able to run a wide variety of analysis tools, meaning the most common I/O patterns of such tools should be supported by the component, enabling easy adaption and addition of new tools to the pipeline. The ability to swap out tools in the analysis pipeline is crucial to facilitate the integration of new and improved biological analysis tools as they are released, and enables the component to follow the evolution within the bioinformatics field, and not get stuck in the past with a hard-coded pipeline developed for specific tools only. E.g., if a tool is discovered not to scale well, it is not a problem the system can remedy, however, having the ability to swap out tools when the tools in use are deemed insufficient is imperative, really establishing the importance of being able to easily adapt to – and add – new tools.

1.3 Requirements

Based on the aforestated challenges, we conjecture that an ideal solution for a big data analysis framework executing workflows of third-party unmodified programs in parallel at scale should satisfy the following requirements:

1. **Scalability:** It must scale gracefully in response to increasing volumes of work and amount of stages. The component should be able to accommo-

date terabyte-scale workloads without incurring substantial unproductive overhead and exorbitant delays.

2. **Genericity:** It should expose an interface that ameliorates extensibility and abstracts away details, in order for the processing component to be easily adaptable to new use cases and for effortless addition of new tools whilst keeping code DRY.

3. **Validation:** The component must be able to determine at time of initialization, prior to beginning a job, whether any of the tools used during the entirety of the workflow to be run are faulty, e.g., the tool not being installed or missing dependencies. Identifying missing tools at an early stage by self-validation can be time-saving, as opposed to discovering it upon invocation at the prearranged workflow stage.

4. **Recovery:** Re-computation of workflow stages should be avoided by any means practicable through aggressively seeking to rebuild state of prior stages whenever applicable. In an effort to diminish time spent debugging and enable jobs to be resumed upon restarting them, results from the last completed stage antecedent to crashing should be restored, given a re-run of a job with unchanged input data following events of failure.

5. **I/O Management:** The component must facilitate the integration of unmodified third-party tools with diverse internal I/O requisites and patterns, including tools that use the standard streams, requires named files for input and output purposes, and combinations of these. It should address file system manipulation, structuring, and maintenance of paths on local and distributed file systems, involving suppression of complexity through handling temporary directories, files, input required and output produced by the tools used.

6. **Idiomaticity:** The internal behavior of the component should reflect that of the underlying framework. Accordingly, using the Spark framework as an example, its behavior should be resembled by forcing a tool, viz., a given function with a corresponding subprocess, to mimic Spark's way of mapping to and from resilient distributed datasets (RDDs). Conforming to these norms promotes ease of use for application programmers already familiar with a given processing framework and its environment.

To the best of our knowledge there exists no system that satisfies all the requirements stated in the preceding.

Existing systems for distributed omics anaylsis include CloudBurst [18] (§ 6.2.3) and Crossbow [19] (§ 6.2.4), which are both implemented using Hadoop MapRe-

duce, limiting their execution flow to a series of map and reduce tasks operating on data tuples, whereas the ADAM [20] (§ 6.2.2) framework and our implementation utilize Spark, offering a more extensible programming model supporting a richer set of dataset transformations and actions. However, both CloudBurst and Crossbow showed promising results for ad hoc parallelization of external programs, but they do not supply a generic framework for addition of new unmodified tools and workflows. Moreover, Crossbow requires modified program binaries for Hadoop integration and preprocessed input files, and ADAM uses novel formats and schemata, currently limited to the field of genomics, meaning any metagenomics tools not directly analogous to their supported genomics functionality would need to be reimplemented in Spark and Scala adhering to their constraints.

1.4 Proposed Solution

This thesis presents COMBUSTI/O, a backend processing component proposed to power the new META-pipe, which includes a workflow manager and abstractions for generation and realization of workflows. COMBUSTI/O is a cloud and HPC compatible framework for parallel pipelined execution of unmodified third-party program binaries exhibiting common I/O patterns designed to handle data at scale. It is built on top of well-established, scalable, and fault tolerant big data systems, leveraging these for distributed input partitioning, dissemination, execution, and aggregation of results. Prior to running the pipeline, a user is required to specify a workflow and implement small tool wrappers with the support of our abstractions, resulting in modest amounts of code (our wrapper implementations ranges from 25 to 108 lines of code) while offering great flexibility as well as mechanisms for stage and task recovery.

The etymological construction of COMBUSTI/O is blending the English word “combust” and the abbreviation “I/O” as a morpheme suffix; the prefix “combust” is a reference to it being implemented on top of the Spark framework, and the “I/O” postfix is appropriate due to it performing a lot of I/O management. This combination forms – the initially anticipated portmanteau – “COMBUSTI/O”, however, “combustio” is an authentic Latin word: The singular nominative “combustio” in Latin cleverly translates to “burning” in English.

COMBUSTI/O satisfies the foregoing requirements as follows:

1. **Scalability:** It is powered by the Spark [21, 22] big data processing engine, running on top of the Hadoop [23] cluster computing framework. Spark is the current leading edge of big data processing, is built to interoperate with Hadoop, and is in active development [24]. This software

stack supports a rich set of operations, easing tool execution and handling, distributed data management, logging, and parallelization, as well as being inherently fault tolerant. Development on top of an already-existing framework displaying these features aids the amendment of the aforementioned shortcomings of the backend processing component of META-pipe 1.0.

2. **Genericity:** As the new architecture is effectuated using the frameworks mentioned in the prior, the component is suitable for both HPC and cloud computing platforms, and is internally comprised of a workflow manager bundled with flexible abstractions masking complex details easing the wrapping and addition of new tools.

3. **Validation:** Prior to execution, a test-run of the tool specified in a tool wrapper parametrized with a help-string is invoked to determine its existence and correct configuration with regard to dependencies.

4. **Recovery:** Task results are stored to local disk upon running a tool, and through persisting stage results to HDFS, state may be recovered at two different granularities. Hence, task results may be recovered from local disk in an attempt to rebuild a stage, or the state may be recovered in its entirety by loading the stage results from HDFS.

5. **I/O Management:** Local and distributed file systems are manipulated by generating an organized directory structure for each workflow and tool that is used locally for input and output of tools, on HDFS when persisting stage results, and is used in the recovery process.

6. **Idiomaticity:** We impose an unenforced convention in which wrappers for tools are written to be independent of the Spark framework. By not convoluting the wrappers with Spark code, each wrapper may be executed from the workflow manager enclosed in the familiar transformations for input partitioning, dissemination, and parallel execution.

1.5 Contributions

To reiterate, in terms of the NIST category definitions, COMBUSTI/O is a platform as a service (PaaS) that can be embedded in a SaaS to be run on top of some IaaS; our real-world use case demonstration integrates COMBUSTI/O as a PaaS, using a subset of the tools deployed in META-pipe 1.0, compatible for incorporation with the new META-pipe SaaS, running on top of a cluster simulating an IaaS.

We explore the feasibility and potency of COMBUSTI/O by providing a demonstration for the marine metagenomics domain, evaluated by applying it to analyze metagenomics samples of raw DNA sequencing data, representing a compute-intensive pipeline, as well as evaluating its utility in data-intensive and latency-sensitive applications by implementing a three-stage workflow consisting of `cat`, `grep`, and `wc`, and a mirrored workflow implementation wrapping Scala built-in methods, correspondingly using small and large datasets, to ascertain its performance characteristics. We also provide best-case measurements using a pragmatic, strictly in-memory, Spark implementation on the same datasets.

Our evaluations show that COMBUSTI/O can facilitate our bioinformatics use case workflow and is applicable for compute-intensive and – depending on accepted latency thresholds – can be used for latency-sensitive workflows. Our results for data-intensive workflows shows that the throughput of COMBUSTI/O is moderate, which is the product of the I/O pattern imposed on it favorable to enabling execution of tools requiring named input and output files. The subprocess forking for remote tool execution is shown to not incur large overheads, and our coarse-grained recovery mechanism is best fit for compute-intensive use, and for testing and debugging purposes of data-intensive workflows where the anticipated number of failures exceed the threshold wherein the time spent recomputing surpasses the combined cost of having the recovery mechanism enabled and rebuilding state.

In sum, the primary contributions of this thesis are:

1. COMBUSTI/O: Abstractions facilitating parallel execution of unmodified program binaries with common I/O patterns for workflow generation, management, and realization using the Spark framework

2. Evaluation of COMBUSTI/O using synthetic workflows implemented in COMBUSTI/O, wrapping binaries and Scala built-ins run on arbitrary datasets to ascertain its propriety for data-intensive and low-latency use cases

3. Demonstration and evaluation of COMBUSTI/O in a real-world use case, namely as the backend processing component of a compute-intensive marine metagenomics analysis service

COMBUSTI/O is a general workflow creation framework supporting ad hoc parallelization of program binaries exhibiting common I/O patterns, facilitating distributed pipelined parallel execution using Spark, while supporting recovery on task and stage level. We conclude, based on our evaluations, that COMBUSTI/O is applicable for compute-intensive, data-intensive and latency-

sensitive applications, that COMBUSTI/O is flexible enough to express complex workflows involving a variety of tools, and that it is not limited to the field of metagenomics nor bioinformatics, but can be used to express any scientific workflow in which distributed and parallel pipelined execution of sequential programs is beneficial. Our framework is best suited for compute-intensive workflows, for which it was originally designed, and we argue its ease of use based on the modest amount of lines of code required for implementing tools and workflows.

1.6 Outline

The remainder of the thesis is organized as follows: Chapter 2 presents the internal and external architectural traits of COMBUSTI/O, as well as an overview of its role in the new META-pipe architecture; Chapter 3 covers the design of COMBUSTI/O, its internal components, including examples of said components, as well as describing the framework stack utilized in COMBUSTI/O; Chapter 4 elaborates on the marine metagenomics use case implemented in COMBUSTI/O, the context of it, and the bioinformatics tool wrappers constituting the workflow; Chapter 5 evaluates COMBUSTI/O, listing the evaluation setup used and measuring end-to-end wall-clock time consumed for data-intensive, latency-sensitive, and compute-intensive workflows; Chapter 6 contains relevant related work, including the frameworks utilized in COMBUSTI/O, basal biological terms, bioinformatics tools wrapped, and bioinformatics pipelines; Chapter 7 concludes; and Chapter 8 discusses possible optimizations and future work.

2 Architecture

This chapter presents both the external and internal architectures of COMBUSTI/O, including its role in the new META-pipe big biological data analysis service, as well as brief summaries of all the components and their interactions.

In short, COMBUSTI/O is an embedded program execution and I/O management framework running on top of horizontally scalable big data storage and processing frameworks supporting the directed acyclic graph (DAG) execution model. It facilitates parallel execution of unmodified program binaries in a distributed fashion by partitioning an input dataset, evenly disseminating the partitions across all participating nodes, and then performing the program execution on each input split. This effectively accommodates parallel execution of inherently sequential programs—presupposing the programs to be executed are data parallel, which is the case for many common bioinformatics tools in use today. There also exists programs natively supporting parallel execution using different frameworks, e.g., the Ray de novo assembler utilizes the Message Passing Interface (MPI), the InterProScan 5 bioinformatic analytics framework uses the Java Message Service (JMS) to distribute workloads, and ADAM uses Spark to distribute and execute in parallel their sorting, duplicate removal, local realignment, and base quality score recalibration pipeline stages, but many still are sequential and data parallel.

We first describe and illustrate COMBUSTI/O's external stack, elaborating on the comprising layers, followed by an overview of its abstractions and internal

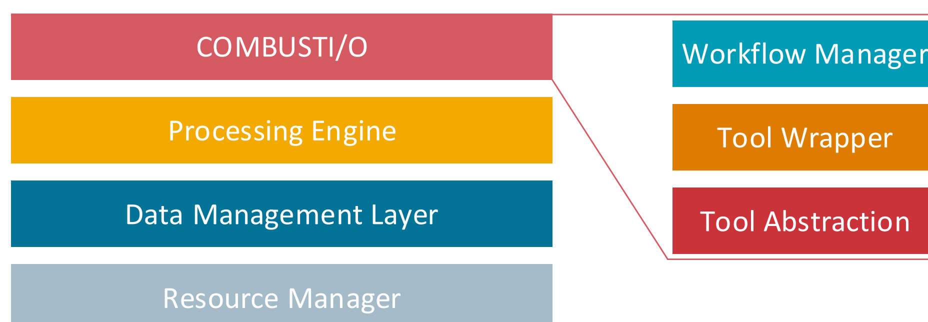

Figure 2.1: Architectural components of COMBUSTI/O

architecture. Finally, a high-level architecture overview of the new META-pipe service in its entirety is outlined to contextualize COMBUSTI/O. The terms "program" and "tool" are henceforth used interchangeably.

2.1 The Architecture Stack of COMBUSTI/O

In favor of separating concerns, the stack used in COMBUSTI/O is decomposed into three layers, in which each layer is represented by a framework capable of efficiently carrying out its obligations on big data in a distributed environment. The COMBUSTI/O stack consists of a processing engine, a data management layer, and a resource manager (Figure 2.1).

COMBUSTI/O COMBUSTI/O is a slim library implemented on top of a distributed dataflow processing framework, interfacing with a data management layer for handling of distributed data and storage, as well as relying on a compatible resource manager for compute and memory resource allotment in distributed environments. It consists of the necessary functionality for distributed parallel execution of tools and aggregation of their results.

Processing Engine The processing engine is a distributed dataflow system supporting a rich set of operators for input data partitioning and parallel execution of DAG workflows, able to persist node failures and endure stragglers. The processing engine assists in the administration of temporary files, including reading, writing, and moving of data, both distributed and local, as well as being suitable for both compute- and data-intensive applications, and provides rich support for additional tools and expansions.

Data Management Layer The data management layer supports cluster-wide distributed storage, is compatible with the processing engine, and tolerates failures of subsets of nodes in a cluster. Performance is key to reduce latencies when performing data-intensive computations, as I/O overhead is costly, and different technologies may be used in combination as there is no exclusivity imposed on this layer.

Resource Manager The resource manager is obligated with efficient resource arbitration and scheduling as required by the processing engine and COMBUSTI/O, able to sustain partial cluster failures.

The architectonic decisions made regarding COMBUSTI/O's external stack was largely influenced by the experiences developing and maintaining META-pipe 1.0. Recall that META-pipe 1.0 is a script-based pipeline framework implemented from scratch, primarily in Perl and UNIX shell, capable of running on HPC platforms. Based on the knowledge obtained through working with this system, we identified that two major disadvantages of utilizing scripting languages for pipeline implementations are failure handling and provenance management.

Common causes leading to crashes and interruption of pipelines include defects in the programs wrapped in the pipeline, corruption of files, disk trouble, and network failures. Scripts are generally not very robust and typically have no functionality for restarting and continuing workflows upon interruption, rather requiring all of the workflow stages to be recomputed regardless of its progress prior to crashing [25]. Disregarding scripts, another option is to use the UNIX Make build automation tool to represent workflows implicitly as dependency trees of the stages to be computed, partially solving the issue of not having to recompute all stages upon restarting a failed workflow [25].

However, neither scripts nor the Make utility have native mechanisms addressing distributed computing, which necessitates implementing this manually. Distributed computing is complex, hard to do correctly, and will increase both code base, and with it, the likelihood of erroneous execution flow. Building on top of well-established big data analytics frameworks, we can rely on these to deterministically handle the distributed aspects of the execution. When dealing with big data, utilizing robust and heavyweight frameworks designed for scalability is advantageous, if not necessary.

The most relevant recent big data system for our use case is ADAM [20, 26], which is a scalable data analytics framework in its own right, built on top of the Spark processing engine, using Parquet [27] and Avro [28] for data representation. They introduce novel custom genomics formats and has reimplemented

various genomics tools to better utilize data access and parallel execution, arguing that legacy formats are ineffective as they were designed with sequential execution in mind. ADAM supports fault tolerance through the frameworks on top of which it is implemented, but due to the custom nature of both formats and tools, in order to use this framework, not only would we have to reimplement in Scala both the metagenomic tools and formats to adhere to their conventions and schemata, but more importantly it breaks with our core principle of utilizing existing unmodified tools whenever feasible, regardless of the implementational details.

In consideration of this, COMBUSTI/O is also integrated with, and exploits the fault tolerant design of, big data analytics frameworks, leveraging their mechanisms to efficiently handle failures and error recovery. Thusly, its stack consists of widely used scalable big data frameworks for processing that supports the DAG workflow execution model, eases data management, and supports a shared file system, as well as having the capability of cloud IaaS platform and HPC environment deployment.

2.2 Internal Components of COMBUSTI/O

The bottom-up design of COMBUSTI/O consists of three main components: the tool abstraction, the tool wrapper, and the workflow manager (Figure 2.2). An application programmer uses the tool abstraction to write tool wrappers for the tools needed for a given workflow, and then writes the workflow to be executed in the workflow manager using said tool wrappers.

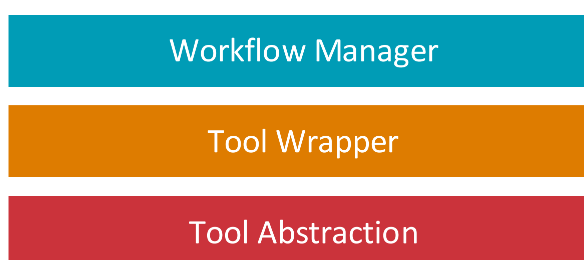

Figure 2.2: Internal components of COMBUSTI/O

Tool Abstraction The tool abstraction supplies functionality necessary for running a program and handling its associated data at the local level, including local recovery mechanisms, suppressing unnecessary details, and is used by an application programmer as a template for implementing new tool wrappers, making the process of adding new tools straightforward. The abstraction can

be seen as the bridge between a tool wrapper and the respective tool to be executed.

Tool Wrapper A tool wrapper consists of the logic required to validate, execute, and retrieve output of the tool that is wrapped. It relies on the tool abstraction to ease the addition of new tools, requiring only the tool-specific code to be added, making use of the flexible interface exposed by the tool abstraction. Tool-specific code typically involves different paths, execution options and flags, and parsers.

Workflow Manager The workflow manager is where workflows are specified by describing a preferred sequence of tool wrappers and connecting the inputs and outputs of the tool wrappers in corresponding order. The parallelization is also handled here, if a tool wrapper implements a data parallel tool, the input is split and sent to participating nodes for execution. Recovery at coarser granularity, at the distributed file system (DFS) level, is also supported in the context of the workflow manager.

2.3 The New META-pipe Architecture

The following describes the envisioned architecture of the new META-pipe, which is the entire big biological data analysis service within which the contribution of this thesis, COMBUSTI/O, makes up an important part, and is intended to serve as the backend processing framework. Development of the new META-pipe is an effort undertaken by the Center for Bioinformatics (SfB) as part of the ELIXIR EXCELERATE WP6: Use Case - Marine metagenomics.

The architectonic components of the complete service are divided into two subgroups: those residing in the external environment and those residing in the execution environment; the former, including a GUI and Web services, being external relative to the execution environment, which refers to the environment within the cluster to be employed for doing the data processing. The external environment consists of a representational state transfer (REST) interface and an adjoining Web service, an object store, and an external manager, and the execution environment consists of an execution manager and a cluster scheduler, namely COMBUSTI/O. A high-level architectonic overview of the biological pipeline service is depicted in Figure 2.3.

Web Site The GUI part of the frontend, used for interactively creating workflows and submitting jobs. It is developed using Node.js [29] – an event-driven framework designed with scalability in mind – and interacts with the exposed

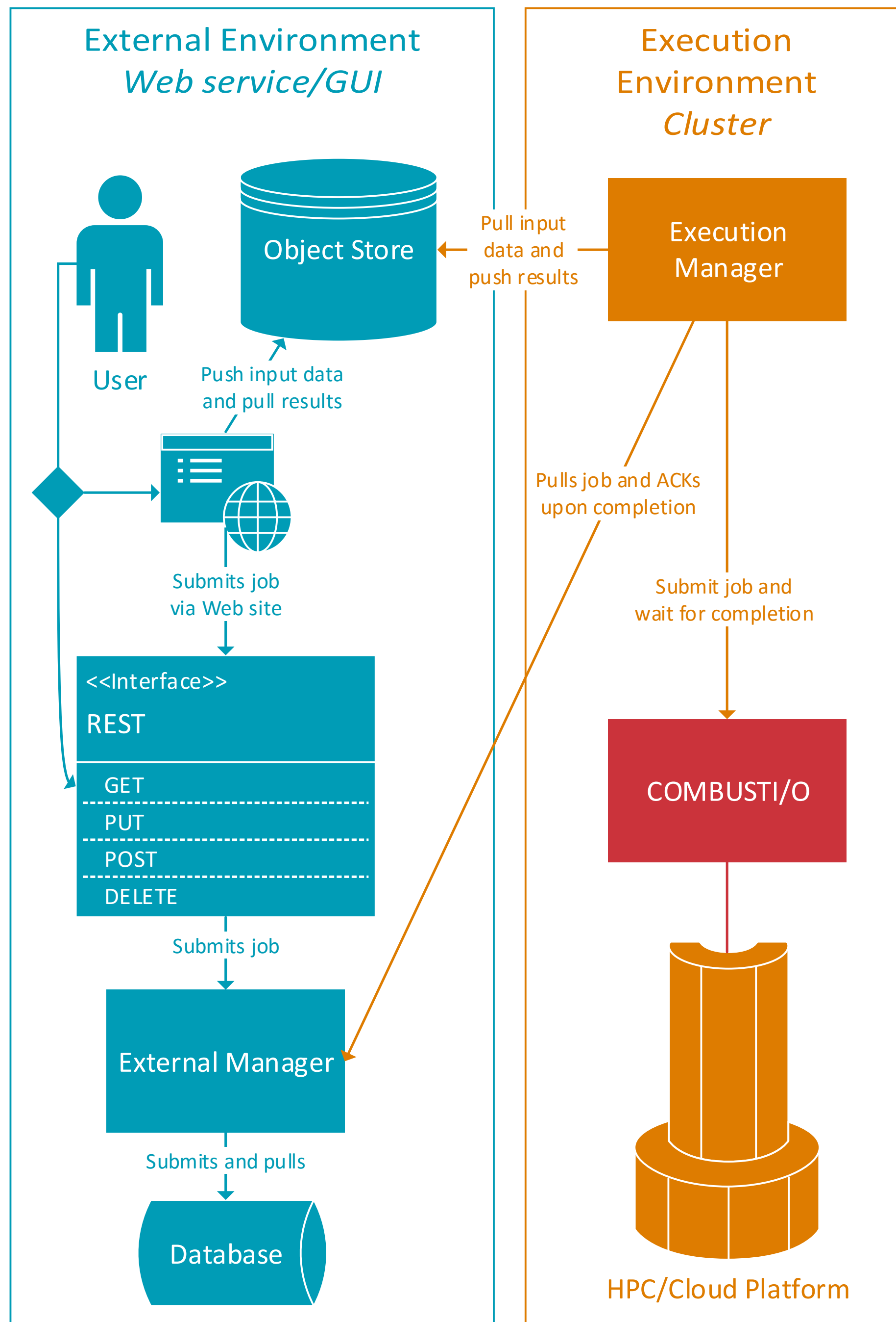

Figure 2.3: Architecture of the new META-pipe biological pipeline service

REST interface by forwarding it job descriptions.

REST Interface The REST interface consists of a subset of the standard RESTful application programming interface (API) HTTP calls, implementing the GET, PUT, POST, and DELETE methods. It submits job descriptions to the external manager and pushes input data to be processed to the object store and waits for notifications from the object store signifying job completion, then pulls the results, making them available for download.

Object Store The object store is a large data store managing and storing input data of jobs to be executed, in addition to storing the corresponding results of the jobs subsequent to fulfillment. It receives input data for jobs pushed by the REST interface and makes it available for transfer to the execution manager. The execution manager later supplies the results of the job for which it priorly pulled input data.

External Manager Keeping track of job descriptions and securely storing them in a database for provenance purposes is the responsibility of the external manager. It accepts job descriptions from the REST interface, writes them to its underlying database, and exposes them for the execution manager to fetch, then waits for an ACK indicating job completion.

Execution Manager Within the execution environment, the execution manager is in charge of orchestrating the appropriate actions compulsory to carrying out individual jobs as they are described. First, it fetches a job description from the external manager and interprets it, followed by pulling the complementary input data from the external environment to the execution environment. Next, it submits the job to COMBUSTI/O for execution, waits for the job to complete, and thereupon pushes the results of the job to the object store, after which it notifies the external manager that the job has been executed by sending an ACK.

COMBUSTI/O COMBUSTI/O is a cluster scheduler that does the actual execution of jobs as they are described, and as such is the processing component of the service. Upon receiving a job description from the execution manager, it simply executes the job.

3 Design and Implementation

This chapter covers the design and implementation of COMBUSTI/O, beginning with an elaboration of the internal design and implementational details, including an example pipeline, followed by descriptions of the tool abstraction, tool wrapper and workflow manager, and finally elucidating upon the framework stack on top of which it is implemented.

3.1 Internal Design of COMBUSTI/O

The design is of utmost importance for the internal elements of the processing component, especially so in pursuance of satisfying the requirements as stated in the introduction. Many design choices are also likely to affect partially coinciding requirements, thus decisions were carefully made by taking priorities into account, emphasizing prominent requirements, yet still attempting to balance the trade-offs. Several design iterations and refactoring cycles were conducted to optimize adherence to the imposed requirements to best fit our predilections, streamline tool wrapper creation, and conform to the "don't repeat yourself (DRY)" principle.

We continue exercising the core design principle of META-pipe 1.0, stating that

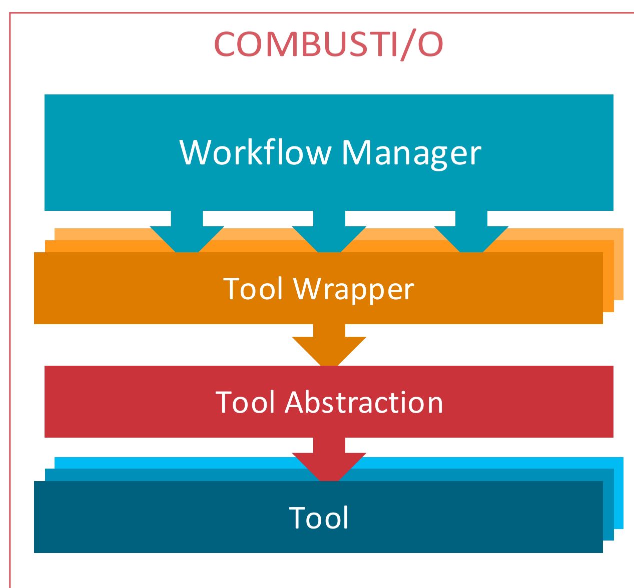

Figure 3.1: Top-down view of COMBUSTI/O's design

existing frameworks and infrastructure services should be utilized whenever practicable. This includes the tools to be used, as one of the principal design goals of ours is for the abstractions to facilitate the addition and execution of any analysis tool that a scientist may provide, relating to compliance with the genericity, compatibility, and I/O handling requirements. One of the major associated drawbacks is having to conform to legacy programs and accompanying formats, as well as requiring additional functionality for conversion of formats in between stages and interpretation of results, and having to write data to disk prior to executing tools and reading result data from disk after, incurring performance penalties in the form of disk I/O overheads.

COMBUSTI/O is configuration based with an explicit paradigm, and schedules work in a per-workflow-manner, consuming an entire workflow at a time. At the very highest level, COMBUSTI/O splits and distributes a dataset, handles I/O reads and writes, and forks subprocesses to execute programs – i.e., UNIX `popen()` – on several machines in parallel, beneficial to reducing wall-clock time spent on pipelined execution of data parallel programs. The interaction of its internal components is shown in Figure 3.1. The ensuing subsections exemplifies a workflow, followed by bottom-up detailed descriptions of the internal components constituting COMBUSTI/O.

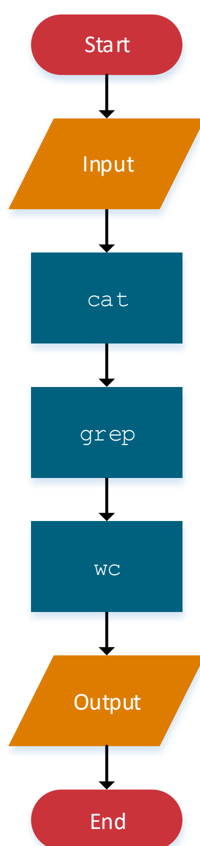

Figure 3.2: Example workflow stages

3.1.1 Evaluation Workflow

In order to evaluate COMBUSTI/O as a workflow manager for data-intensive and latency-sensitive applications, we created a pipeline (Figure 3.2) to be replicated using our abstractions, exemplified as a UNIX shell command in Code Listing 3.1. It is a three-stage pipeline consisting of the common UNIX tools `cat`, `grep`, and `wc`, respectively used to write, search, and count words.

Code Listing 3.1: UNIX shell evaluation pipeline

```
cat path/to/file.txt | grep "query" | wc -w
```

The UNIX `pipe()` uses the standard output of the preceding command as standard input for the following, meaning the inputs and outputs only reside in memory during execution. This behavior can be easily emulated using the native Spark and Scala APIs, as shown in Code Listing 3.2. Here, the `textFile` method reads a text file into an RDD of lines, mimicking the `cat` tool; `filter`

mimics grep's functionality; and the map transforms the RDD to represent each line as an integer of the number of words it contains, followed by a reduce that sums the integers using arithmetic addition, which altogether mimics wc -w.

Code Listing 3.2: Pragmatic Spark evaluation pipeline implementation

```
sparkContext.textFile("path/to/file.txt")
  .filter(_.contains("query"))
  .map(_.split(" ").count(_.nonEmpty))
  .reduce(_ + _)
```

However, COMBUSTI/O should provide provenance and support the common I/O patterns of program binaries, making the implementation more complicated than conveyed by the Spark example above. For both provenance management and to facilitate the execution of programs requiring named input and output files, input and output of each tool needs to be stored to disk. Because of this, Code Listing 3.1 does not accurately reflect the requirements imposed upon COMBUSTI/O.

A shell pipeline representative of what COMBUSTI/O performs is shown in Code Listing 3.3. As inferred from the code, the standard error and output streams are written to files, and the output file of the previous stage orchestrated to serve as the standard input of the following stage, instead of being directly piped using memory only. Note that COMBUSTI/O does not support programmatic redirection of the standard input stream, but rather use named files as command line arguments for input purposes.

Code Listing 3.3: Accurate UNIX shell evaluation pipeline

```
cat  < path/to/file.txt 2> catErr  > catOut  ;
grep < catOut  "query"  2> grepErr > grepOut ;
wc   < grepout -w       2> wcErr   > wcOut
```

In order to implement this workflow in COMBUSTI/O, a tool wrapper for each unique tool (cat, grep, wc) is required. Using these tools, a workflow can be specified from within the workflow manager, as shown in Code Listing 3.4. The code is curtailed for brevity, mainly omitting details including user-specified arguments, initializations, and the coarse-grained recovery mechanism, as well as parallelization techniques. In order to parallelize the execution, the tool wrappers are enclosed in a map transformation using Spark, and is exemplified in a later section.

Note that the redirection of standard output and error streams are done in the context of the tool wrapper, not the workflow manager, hence, it is not reflected in the code of Code Listing 3.4.

Code Listing 3.4: Example workflow implementation of the example UNIX pipeline. The code is abridged for concision

```
class WorkflowManager(context: => SparkContext) extends Command  {
                                   ⋮

  override def apply(): Unit = {
    val localOutputPath = "/tmp/"
    val jobPath = buildJobPath(localOutputPath, "myJobId")

    // Run cat on input file
    val catPath = buildToolPath("cat", jobPath)
    val catInput: CatInput = CatInput("file.txt")
    val catContext = new ToolContext {
      def help:         String  = "--help"
      def program:      String  = "/bin/cat"
      def path:         String  = catPath
    }

    val catWrapper = new ToolWrapperImpl(new Cat)
    val cat: CatInput => CatOutput = catWrapper(catContext)
    val catOutput: CatOutput = cat(catInput)

    // Run grep on catOutput
    val grepPath = buildToolPath("grep", jobPath)
    val grepInput: GrepInput = GrepInput("query", catOutput)
    val grepContext = new ToolContext {
      def help:         String  = "--help"
      def program:      String  = "/bin/grep"
      def path:         String  = grepPath
    }

    val grepWrapper = new ToolWrapperImpl(new Grep)
    val grep: GrepInput => GrepOutput = grepWrapper(grepContext)
    val grepOutput: GrepOutput = grep(grepInput)

    // Run wc on grepOutput
    val wcPath = buildToolPath("wc", jobPath)
    val wcInput: WcInput = WcInput(grepOutput)
    val wcContext = new ToolContext {
      def help:         String  = "--help"
      def program:      String  = "/usr/bin/wc"
      def path:         String  = wcPath
    }

    val wcWrapper = new ToolWrapperImpl(new Wc)
    val wc: WcInput => WcOutput = wcWrapper(wcContext)
    val wcOutput: WcOutput = wc(wcInput)

                                   ⋮

  }
}
```

3.1.2 Tool Abstraction

The tool abstraction implements functionality for easing integration of new tool wrappers, in particular facilitating the forking of subprocesses executing program binaries and accompanying arguments, redirection of output and error streams of the program, and creation and management of local per-tool directory structures for input and output purposes (Figure 3.3).

A helper-function for execution of programs lets the user redirect `stdout` and `stderr` to files by supplying destination file paths and setting a flag, along with the specified program, its arguments, and path to the execution environment. Executors on a common node share job, stage, and tool directories; collisions are avoided using uniquely indexed file names.

An interface of four functions (Code Listing 3.5) is exposed to be used by a tool wrapper implementor method to perform the execution logic, which we found to be the most pliable set of functions for conducting said logic, while seeking to mask superfluous details. The first two functions (`validateBefore` and `recoverable`) contains implementations for the common case, but may be overridden for special cases; the remaining two (`execute` and `output`) are implemented by the user for each tool wrapper, and consists of facilitating the input and output requirements of the tool (`execute`), and reading, parsing, and returning output (`output`).

Code Listing 3.5: Execution logic interface

```
def validateBefore(help: String)
def recoverable(index: Int, uId: String): Boolean
def execute(): Int
def output: Out
```

Our original design intended to implement and expose two functions, `prepare` and `command`, to constitute the execution logic, but was however replaced in favor of a unitary `execute` function to coalesce the execution implementation and provide more flexibility.

The validation method in use validates a tool by performing a test-run using a help-string and asserting its successful exit status. If a tool does not implement a help-string, the test-run is performed in order to ascertain its presence by asserting that its exit status does not signify the file not being found.

Recovery at the local level is complex when used in distributed environments due to Spark on Yet Another Resource Negotiator (YARN) not deterministically choosing nodes on which to deploy executors, i.e., the same node is not guaranteed to host a given executor. Moreover, the distribution pattern of partitions

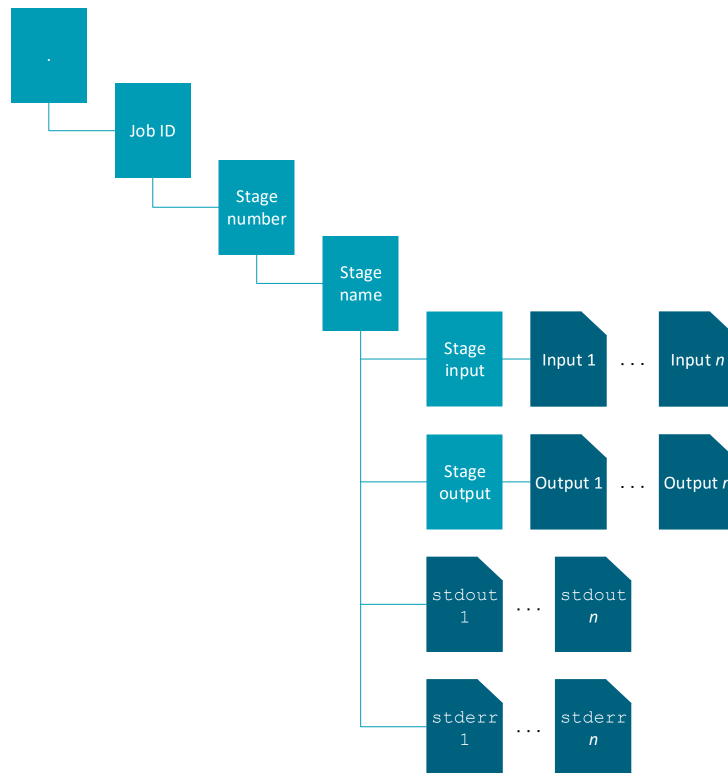

Figure 3.3: Directory structure created per tool and stage in the pipeline by the tool abstraction with user-specified names

vary, thus assumptions cannot be made with regard to each partition being sent to the same executor. Our recovery mechanism tests whether an output file indexed with the assigned partition is present, if the given run was successful, and if it matches a per-workflow ID specified by the user, and only then is the output recovered.

This ID is specified prior to running as a command line argument to the application, and it is important to change the ID when re-running a workflow with a different amount of executors, as then the partitions of the previous run will not match those of the current, and may result in the recovery of erroneous output.

3.1.3 Tool Wrapper

A tool wrapper is written by the user, and relies on the implemented functionality of the tool abstraction, and contains the tool-specific code. It inherits the

Code Listing 3.6: Tool wrapper implementor execution logic

```
if(!recoverable(context.index, context.uId)) {
  validateBefore(context.help)
  execute()
}

output
```

functionality by extending the tool factory, which implements the tool abstraction, enabling the use of its entire interface.

The `execute` function is used to facilitate the execution of the tool wrapped, and will commonly consist of writing input to be processed to disk, directing output streams, and parametrization and execution of the tool. The `output` function typically reads produced output from disk to memory, converts the format using some parser, and returns the resulting dataset.

A tool wrapper implementor performs the fixed execution logic of the implemented interface of the tool wrapper, consisting of the four functions as exposed by the tool abstraction. Code Listing 3.6 shows the logic, in which the hypothetically existing output of the assigned partition is investigated, and if found, the `output` method is invoked directly; else the tool is validated then executed on the partition prior to invoking `output` method.

3.1.4 Workflow Manager

It is in the workflow manager a workflow constituting stages is arranged to the developer's preference and are executed in sequence as specified, consisting of one or more stages. Each stage involves specifying input and arguments to a tool wrapper, which is dispatched upon invocation of the corresponding tool wrapper, and consequent to stage completion, the output is available for further use by the application programmer from within the workflow manager.

A stage may be executed once on the master node, or the execution may be distributed and run in parallel on several nodes, given a data parallel tool. Spark code is confined to the workflow manager to keep the lower level abstractions less convoluted and to sanction a concept in which each tool wrapper is to be run once per input or input partition; an aspect we find appealing as it may help application programmers reason about code and choices when writing tool wrappers and workflows, conjointly maintaining code readability.

Using the Spark API, input may be partitioned and each partition processed

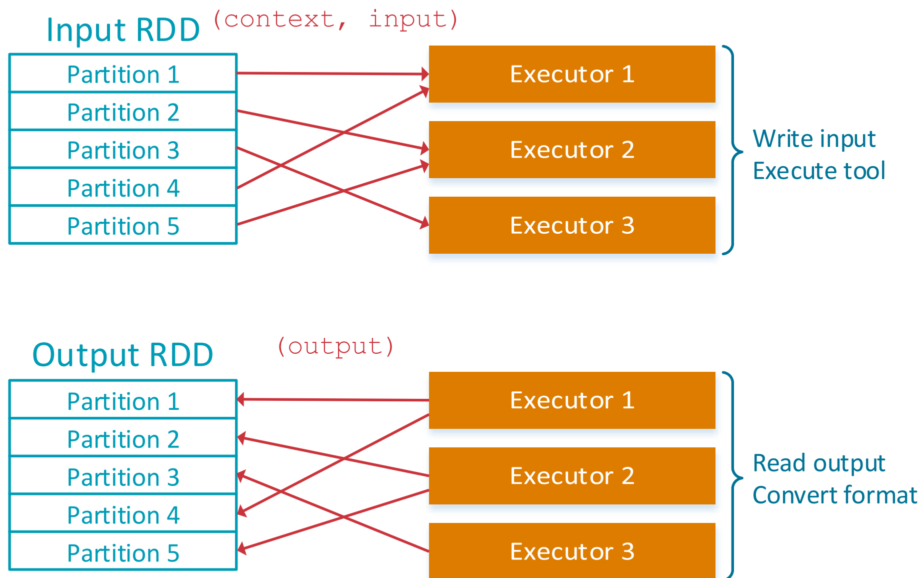

Figure 3.4: Scheduling, dissemination, and execution using Spark

using a tool wrapper by an executor in parallel to leverage both data parallelism and locality, and to distribute the workload. Figure 3.4 shows a simplified example of input partitioning and dissemination, the execution flow, and how output is mapped back to an output RDD using an input RDD with 5 partitions and 3 executors. If a program utilizes some external parallelization mechanism, it may be invoked once by the master (driver) node.

The optimal granularity at which to repartition, distribute, and disseminate input data mainly depends on the characteristics of the tool and the size of the input dataset, but is also influenced by several other factors, including hardware features and number of nodes available for processing. Thence, the process of repartitioning datasets is not straightforward as there is likely no optimal static value for datasets, but rather needs to be dynamically explored to obtain the optimum partition size for each tool and complementing wrapper.

Code Listing 3.7 illustrates how the Spark `map` function may be used in the workflow manager to disseminate and execute a tool on each partition of an RDD. The input RDD may be repartitioned using the Spark `repartition()` transformation, and the standard `parallelize()`, `objectFile()`, and `textFile()` methods all use the `defaultParallelism` variable to determine the number of partitions by default, which is set to be the largest of 2 or the aggregate number of cores on all executors.

The recovery at the workflow manager level is implemented using the Spark

Code Listing 3.7: Example map for distributing and disseminating tool execution

```
val out: RDD[ToolOutput] = inputRDD.mapPartitionsWithIndex {
(partitionIndex, partition) =>
  val context = new ToolContext {
    def index: Int = partitionIndex
    def help: String = "--help"
    def program: String = "/bin/tool"
    def path: String = toolPath
    def uId: String = uniqueId
  }

  val toolWrapper = new ToolWrapperImpl(new Tool)
  val tool: ToolInput => ToolOutput = toolWrapper(context)

  Iterator(tool(ToolInput(partition)))
}
```

`saveAsObjectfile()` function to persist stage results to HDFS as serialized Java objects. The execution logic follows that of the tool abstraction, i.e., deciding whether or not to recompute a given stage based on output file presence, and is the primary recovery mechanism, making the tool abstraction recovery a contingency option (fallback).

This overaggressive approach to state recovery at the workflow manager level may lead to unnecessary loading of stage output from HDFS, as the state is loaded based only on the predicate concerning the current stage in question, viz., recovering all stages when only the last one was actually required is a possible outcome. To remedy this, a reference containing the last recoverable stage may be saved until an unfinished stage is encountered, then lazily loading only the required stage into memory. This is, however, left as future work.

3.2 Framework Stack of COMBUSTI/O

The COMBUSTI/O backend pipeline architecture is built on top of a best-of-breed software stack consisting of open-source Apache [30] projects, all designed to be run on clusters of commodity machines. The proposed software were carefully chosen to comply with the architectural requirements imposed, and developing under the aegis of well-established big data analytics frameworks has its boons in terms of low-cost fault tolerance and handling, scalability, reliability, and high availability.

The Spark big data processing framework is utilized as the core processing

engine; Hadoop Distributed File System (HDFS) represents the data management layer, and interfaces with Spark, assisted by the Network File System (NFS) for facilitating MPI jobs; and YARN is the cluster resource manager in use (Figure 3.5).

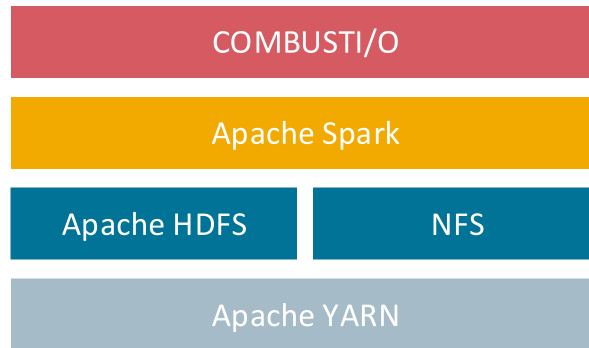

Figure 3.5: The framework stack utilized in COMBUSTI/O

Processing Engine: Spark Among the frameworks considered representing the processing engine were Apache's Hadoop MapReduce [23], Spark [21] and Pig [31], and Microsoft's Prajna [32], Naiad [33] and DryadLINQ [34]. In theory, any processing engine supporting the DAG workflow execution model may be applied, however Spark is the framework in use for the META-pipe project, and was thusly chosen to represent the processing engine of COMBUSTI/O for compatibility reasons. The Spark big data processing engine exposes a high-level Scala API with a rich set of operators for partitioning, distributing, and processing data at scale. Its main advantage over MapReduce systems is aggressively seeking to do in-memory computations using their RDD abstraction and a supplied API for doing parallel operations on these datasets called transformations and actions. Moreover, fault tolerance in Spark is cheap due to only logging the set of transformations required to compute a given RDD, enabling recomputation of RDDs while omitting replication. Spark can be run in standalone mode, on top of Mesos, or on top of YARN, and interfaces well with HDFS as it was designed to enhance the Hadoop stack, rather than to replace it. See § 6.1.2 for elaborate information on Spark.

Data Management Layer: HDFS and NFS Among the considered options were Apache HDFS [35], Amazon Simple Storage Service (S3) [36], Microsoft's Azure Storage [37] and Distributed File System [38], and NFS [39]—although strictly speaking not a clustered file system per se. As the processing engine needs to interface with the data management layer, we decided on HDFS, which is the distributed file system of the Hadoop framework designed for fault tolerant storage and manipulation of big data. It supports aggregate

read performance, consistency, replication, data locality, and exposes a familiar Java API for programmatic file system manipulation, supporting seamless interaction in Spark using Scala. It is used to store input for, and output of, a given workflow's stages, which enables the recovery of serialized stage output. A caveat called to attention by Nothaft et al. [26] is that HDFS is easily scaled out, but more difficult to scale down, giving an incentive to store most data in block storages of cloud service vendors and keep only frequently accessed intermediate files in HDFS. This is, however, beyond the scope of this work, but could be facilitated using the Spark compatible Amazon S3. NFS is used in combination with HDFS to facilitate MPI tasks, and does not require each participating node to have a local copy of input data. For more details on HDFS and NFS, refer to § 6.1.1 and § 6.1.3, respectively.

Resource Manager: YARN Among the candidates for representing the resource manager were Apache's fine-grained Mesos [40] and coarser-grained Hadoop YARN [41], and Microsoft HPC Pack [42]. YARN is used for arbitrating resources for Spark in COMBUSTI/O, and is a natural choice when operating on a Hadoop cluster. It requests and assigns resources at a coarse level of granularity and in a homogeneous fashion, in which each container running an executor is statically assigned a fixed amount of resources which are bound to the container for the duration of its lifetime. It is fault tolerant, scalable, and allows for multiple users to share a common cluster. Refer to § 6.1.1 for more details on YARN.

4

Use Case: Marine Metagenomics

In order to assess the performance and potential of the abstractions and modules constituting COMBUSTI/O in a real-world compute-intensive pipeline, we have implemented a use case for marine metagenomics by wrapping a subset of the biological analysis tools of an already existing big biological data analysis pipeline, META-pipe, and executing said tools in parallel using our abstractions. This chapter presents the workflow implemented, its context, and implementational details with regard to the tools and complementary wrappers; we evaluate this workflow of tools in the following chapter. Readers unfamiliar with elementary biological and bioinformatics concepts are urged to peruse § 6.3 prior to continuing.

4.1 Use Case and META-pipe 1.0

The background for developing META-pipe 1.0 and this work is a use case within the ELIXIR EXCELERATE project, in which a marine metagenomics analysis pipeline is a projected service of the ELIXIR infrastructure.

ELIXIR (§ 6.6) is a European bioinformatics platform providing open-access infrastructure for research in life sciences, and is an international collabora-

tion of assorted bioinformatics institutes and their services; EXCELERATE is a project launched for implementing and supporting operations of ELIXIR services, aiming to efficiently coordinate and govern the European bioinformatics community and supply leading edge life science infrastructure [43]. One of the services of the EXCELERATE project (WP6) involves a marine metagenomics data analysis services use case to be implemented pursuant to establishing gold standard databases, analysis tools and pipelines for said domain, and forms the context of META-pipe 1.0 and this work.

As may be recalled from the introduction, META-pipe 1.0 is the pipeline applied to generate data for insertion in the marine reference database. It supports analysis and annotation of sequence data, both genomic and metagenomic, consists of several tools for pre-processing, taxonomic classification, and functional analysis including visualization, and may be operated using the Galaxy workbench. For further details on META-pipe 1.0, please refer to § 6.2.1.

In the broadest sense, the backend processing of significance in this work consists of three stages, beginning with raw DNA sequencing reads, which are i) assembled, followed by ii) gene prediction on the assembled contigs, and finally iii) annotation of the predicted genes. These stages are analogous to using the tools i) Ray Méta, ii) MGA, iii) BLAST (`blastp`) and the InterProScan 5 suite (using TIGRfam, ProDom, SMART, PROSITE, HAMAP, SUPERFAMILY, PRINTS, PANTHER, Gene3D, PIRSF, COILS, and Phobius).

We argue these tools to be representative of the common cases, by reason of this set of tools covering a wide range of functionality and I/O patterns. As a deduction we claim that, on the grounds that COMBUSTI/O may be used for this set of tools, it may also be used for any other tools exhibiting homologous functionality and I/O patterns.

COMBUSTI/O forms the basis and inspiration for the cluster scheduler backend envisaged for the new META-pipe, and is currently being improved upon and under further development at the SfB.

4.2 Implemented Workflow and Tools Wrapped

The DAG of stages implemented in the workflow for the marine metagenomics use case representing a compute-intensive pipeline is shown in Figure 4.1. The implemented workflow is a slight variation of the original META-pipe, as the original version implements the `blastp` command line tool, whereas this work implements `blastx`; the implication of this is the direct use of nucleotide sequences, relying on `blastx` for translation to peptides, as opposed to manually

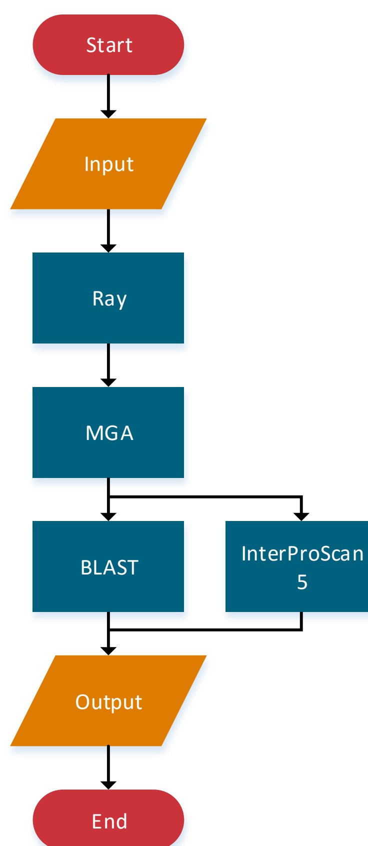

Figure 4.1: Stages of the implemented workflow

performing the translation of the predicted genes and using `blastp`. Not having done the translation manually also impacts the execution of InterProScan 5, as we also here directly use nucleotide sequences, antagonistic to using peptide sequences as done in META-pipe. Furthermore, of the tools utilized in the InterProScan 5 suite, Phobius, is omitted in the workflow implemented using COMBUSTI/O. Thusly, the marine metagenomics use case pipeline has no basis for direct comparison with META-pipe 1.0, but primarily serves as a proof-of-concept.

It is, however, of importance to acknowledge the resulting divergence as not being an impediment of COMBUSTI/O, but a product of choosing the simpler way when implementing the analysis workflow due to time constraints (largely a product of translation issues BioJava4, a desire to be liberated of said library, and the Phobius issue is based on restricted file system privileges). Moreover, these choices are not linked to the performance of the COMBUSTI/O framework per se.

Various parsers and format conversion tools for prevalent bioinformatics formats comes bundled with our abstractions as a utility library, and is part of our abstractions, simplifying I/O handling and reducing development efforts required.

The following elaborates on the tool wrappers implemented for the marine metagenomics pipeline that serves as the use case forming the demonstration of principle and feasibility of COMBUSTI/O.

4.2.1 Ray Wrapper

The Ray wrapper contains functionality facilitating the open-source scalable de novo assembler Ray, more specifically the Ray Méta module, which performs de novo metagenome assembly. The tool itself is highly configurable, supporting a great many command line arguments; the arguments used in the wrapper is listed in Table 4.1. See § 6.4.1 for more details on Ray, and a complete list of arguments and directions to download mirrors can be found on its Website [44].

Since Ray utilizes MPI for distribution and inter-process communication (IPC), the wrapper is designed so as to only be invoked once by the driver, and rely on `mpirun` for parallel execution given the number of processes to launch and a file listing the available hosts.

Table 4.1: Ray arguments

Argument	Explanation
`-k`	Set k-mer length
`-p`	Specify two paired-end read files
`-minimum-contig-length`	Set minimum contig length, in number of nucleotides
`-o`	Specify output file path

The wrapper takes as input the paths of two files containing paired-end reads. Using the `execute` function, the files are copied to NFS prior to running the job using MPI through `mpirun`, the arguments to which is listed in Table 4.2. After completion, the input and output directories residing in NFS are moved to the local disk to the corresponding job directory, and the output produced to `stdout` is scanned for keywords signifying a faulty run, as `mpirun` may exit with a successful exit status regardless of the exit status of Ray.

The `output` function reads, parses, and returns the relevant output, which are

Table 4.2: `mpirun` arguments

Argument	Explanation
`--np`	Number of processes to execute
`--hostfile`	Specify hostfile with list of hosts to run on
`--mca`	Arguments to MCA modules (e.g., ignore an interface)

the FASTA formatted contigs produced.

4.2.2 MGA Wrapper

The MGA wrapper enables the execution of MGA for gene prediction of assembled contigs. MGA only takes one input parameter, which is the FASTA file on which to perform gene prediction and writes its output to `stdout`. The tool is data parallel and may thus be run in parallel, and is openly available on its Website [45]. Refer to § 6.4.2 for further information on MGA.

Its wrapper takes a path as input, and its execution only consists of writing the FASTA file to disk and invoking the tool, redirecting `stdout` to a file.

The `output` function reads and parses the MGA output, obtaining the different fields, followed by reading the original input FASTA file in order to extract the sequences of predicted genes, demarcated with start and stop positions, and optionally reversing the sequence depending on the sign of its strand. Both the raw output of MGA and the derived predicted sequences are returned.

4.2.3 BLAST Wrapper

The BLAST wrapper uses a command-line interface (CLI) application developed at the National Center for Biotechnology Information (NCBI), `blastx`, using translated nucleotide sequences to search protein databases. It is highly configurable and supports many command line arguments, and the arguments set in our wrapper are listed in Table 4.3. § 6.4.3 contains a more detailed description of BLAST; for more information and open-access, visit NCBI's BLAST Websites [46].

By observation, memory usage seems to directly correspond to the size of the searched BLAST database, and when running several tasks in parallel the amount of memory per executor is rapidly exhausted given large databases,

Table 4.3: `blastx` arguments

Argument	Explanation
`-db`	Path to BLAST database
`-query`	FASTA file containing query sequences
`-out`	Specify output file path
`-max_target_seqs`	Number of aligned sequences to report
`-outfmt`	Alignment view format
`-evalue`	Expectation value save threshold
`-dbsize`	Cumulative size of database
`-num_threads`	Number of threads to schedule

unless the nodes of the cluster in use is configured with a large amount of dynamic random-access memory (DRAM) to CPU virtual core ratio. Thus, we support running on a split database in our wrapper; the current configuration searching against the UniRef50 protein database split in 8.

The implication of having several databases is having to tally the sum of nucleobases in all sequences in the source FASTA file used to create the database splits prior to running the wrapper, conducive to acquiring a correct expectation value for the results produced by `blastx`. This number is provided as the database size command line argument to the tool, and is calculated only once at the workflow manager level.

Let R be all records in the FASTA source, S the set of splits, N the number of splits, and s the sequence length, then

$$\sum_{r=1}^{N} S_{r_s} = \sum R_s$$

is the accumulated sequence length calculated prior to running the job, such that the cumulative sequence length of the source FASTA file is equal to the cumulative sequence length of all splits used for database creation. For recovery purposes, this value is also persisted to local disk in context of the workflow manager.

The wrapper takes as input the predicted sequences for querying and the BLAST database size, and executes the tool iteratively using the same queries on each database split in succession, producing additively more unique output files contingent upon the number of splits.

The `output` function iteratively parses, extracts fields, and accumulates the output of all runs prior to returning.

4.2.4 InterProScan 5 Wrapper

InterProScan 5 is a bioinformatics software suite – a shell script invoked using the Bourne-again shell (`bash`) – and its wrapper consists of functionality for running it in standalone mode. InterrProScan 5 encapsulates several other tools, of which a subset are deployed from the wrapper, and are used to query supplementary signature databases for predictive models. Table 4.4 lists the arguments set in the wrapper; for more information on InterProScan 5, please refer to § 6.4.4 or visit the EBI InterProScan 5 Websites [47].

Table 4.4: InterProScan 5 arguments

Argument	Explanation
`-goterms`	Enable gene ontology lookup
`-iprlookup`	Enable InterPro lookup
`-f`	Set format
`--applications`	List analyses to be run
`-t`	Set input type, nucleotide or protein
`-i`	Specify input file path
`-o`	Specify output file path

Its wrapper takes as input the predicted sequences, which are passed on as a command line argument upon launching the script.

The `output` function parses, extracts the fields of the InterProScan 5 format, and returns the results.

4.2.5 Marine Metagenomics Workflow

The workflow implemented incorporates the above tools, connecting the inputs and outputs in sequence such that the output of Ray is used as the input for MGA, and the output of MGA is used as input for both `blastx` and InterProScan 5. Figure 4.2 shows the relationship between the wrappers, abstraction, and tools, as well as the order in which they are executed from the workflow manager.

InterProScan 5 and BLAST reside in the same column to indicate that both are dependent on the output of the previous stage – MGA – and thus may be run concurrently; we implement the wrappers to run in sequence, as we find no

rationale to motivate their simultaneous execution.

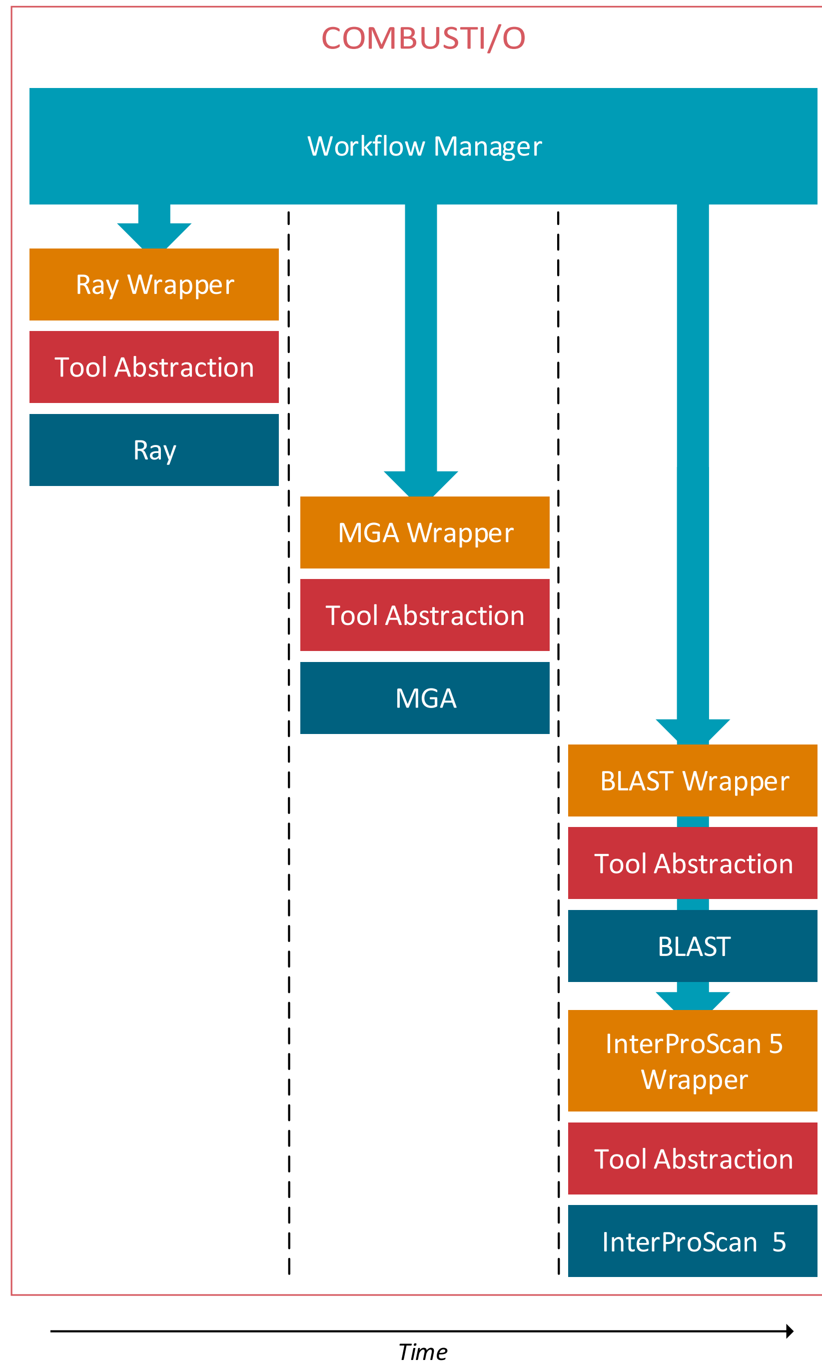

Figure 4.2: Pipelined execution of the stages in the implemented workflow

In more detail, the Ray wrapper takes as input two paired-end reads, generally compressed FASTQ files, produces a number of outputs, but only the assembled FASTA contigs are retrieved; the contigs forms the input of the MGA wrapper, which predicts genes and extracts the predicted sequences from the source subset of the FASTA contigs; the FASTA sequences of predicted genes constitute the input for both `blastx` and InterProScan 5, using them as queries for database searches, and antecedent to completing, the results are parsed and returned in their respective formats.

4.3 Implementation

Ease of use and simplicity are of great appeal to users when choosing whether or not to use a given framework, and keeping code DRY and having functionality for the common case is essential for minimizing user implementation efforts. Accordingly, we want our framework to be intuitive and should save time to use, in comparison to implementing impromptu specialized pipelines.

Measuring ease of use and appeal of such frameworks is not an exact science, and usually breaks down to subjective user surveys. Number of lines of code required to achieve some behavior in an implementation is another, arguably more objective, measure to this end. The tool wrappers implemented in the marine metagenomics use case range from 63 to 108 lines of code, and the workflow manager was the largest of the modules required implemented by the user with its 370 lines. Table 4.5 lists number of lines of code for the workflow manager and each tool wrapper.

It is worth noting that the wrappers for BLAST is a special case, since it executes several times producing multiple output, the local recovery mechanism needed to be overridden to reflect this behavior. We believe these to be modest amounts of lines of code, however, the workflow manager is bloated and refactoring would likely reduce its size.

Table 4.5: Lines of code per module, as reported by the `cloc` tool

Module	Lines of code
`WorkflowManager.scala`	370
`BlastWrapper.scala`	108
`RayWrapper.scala`	97
`InterProScan5Wrapper.scala`	76
`MgaWrapper.scala`	63

The code is open-source and all wrapper implementations may be found on

GitHub (see Appendix A).

5 Evaluation

This chapter presents our evaluation of COMBUSTI/O, demonstrating its applicability by running three distinct workflows, representing compute-intensive, data-intensive, and latency-sensitive applications, measuring end-to-end running times to assess the overhead of COMBUSTI/O, the overhead of its recovery mechanism on the workflow manager, and the cost of remote tool execution.

The recovery mechanism at the workflow manager level may be beneficial for long-running workflows, debugging, and testing purposes, as it allows for restarting workflows at stage granularity by rebuilding the state of the last completed stage. We therefore evaluate differences in running times with it enabled and disabled to assess the overhead incurred for the different workflow flavors.

We begin by describing our methodology, followed by presenting the results, discussions, and conclusions of our evaluation.

5.1 Methodology

This section covers the setup of the cluster used for running experiments, including its hardware and software specifications, succeeded by descriptions of the Spark configuration options used for the different workflows. Follow-

ing this, we describe in detail the workflows and their configurations used for evaluation, and finally we reason about our choice of measurements.

5.1.1 Cluster Specs and Configuration

To evaluate, we use 14 compute nodes of the `ice2` 15 node cluster, comprised of heterogeneous nodes consisting of two different types. Type *A* nodes count 9, are the newest, and the resource composition of these nodes was formed especially with data-intensive computing in mind, with emphasis on a large amount of DRAM, to facilitate the current generation of in-memory frameworks; type *B* nodes count 5 and were also composed to efficiently do data-intensive tasks, with two CPUs per node and originally several disks per node, which is an artifact of the disk-intensive MapReduce era. The hardware specs of the two node types are listed in Table 5.1 and Table 5.2 respectively.

Table 5.1: Hardware specifications of a type *A* `ice2` node

CPU	Intel® Xeon® Processor E5-1620 (10M Cache, 3.60 GHz)
Memory	32 GB
Storage	4 TB
Operating system	CentOS Final (6.7)
Cluster manager	YARN
Network	Ethernet 1 Gbit/s

Table 5.2: Hardware specifications of a type *B* `ice2` node

CPU	2 × Intel® Xeon® Processor E5620 (12M Cache, 2.40 GHz)
Memory	24 GB
Storage	1.5 TB
Operating system	CentOS Final (6.7)
Cluster manager	YARN
Network	Ethernet 1 Gbit/s

Table 5.3 shows cluster metrics for the different node types, the total amount of resources, and resources available to YARN. Note that all numbers exclude the frontend of the `ice2` cluster, and that YARN's ResourceManager runs on a node of its own and sees homogeneous resources (i.e., only 8 virtual cores of the type *B* nodes are available for requisitioning). The configured capacity of the HDFS is 33 TiB with a replication factor of 3 and block sizes of 128 MB. `ice2` is a Cloudera cluster implementing Hadoop version `2.6.0-cdh5.4.9`.

We argue that the `ice2` cluster used for evaluation is representative of our

Table 5.3: Accumulated cluster metrics of `ice2` in sum and metrics as seen by YARN

Perspective	Nodes	Virtual cores	Memory
`ice2` type *A*	9	72	288 GB
`ice2` type *B*	5	80	120 GB
`ice2` total	14	152	408 GB
YARN	14	112	140 GB

target cloud and HPC platforms in terms of the software composition, although equipped with heterogeneous hardware, its environment reflects those of the target platforms. Moreover, it is a dedicated cluster, making it fit for testing latency-sensitive tasks by circumventing queueing mechanisms.

In production, it is likely that the amount of nodes deployed will be larger than the capacity of `ice2`, in order to decrease wall-clock time consumed per workflow. However, our abstractions and tool wrappers are designed to be run independently of one another, imposing no additional scalability bottlenecks, thus we postulate that the question of scalability deflates to being contingent on the scalability of Spark and the underlying software stack.

5.1.2 Spark Configuration

All tests were run using Spark version `1.3.0`, being the latest Cloduera-supported version, and configuration options were set in-line upon submitting applications using the `spark-submit` tool. For more information on the available configuration options regarding Spark configuration, Spark on YARN, and job scheduling, please refer to the documentation [48, 49, 50].

The Spark framework supports an abundance of configuration and tuning options, useful for tweaking application execution; the set of options modified for the evaluation cases are described below. Unless otherwise stated, the remaining options are all set to default values.

An example of a submit-script for utilizing all the YARN resources available on `ice2` is shown in Code Listing 5.1. The following elaborates on the subset of Spark configuration options that were set for evaluating our framework, and Figure 5.1 visualizes their architectural affiliations in the Spark, YARN, and JVM ecosystem:

`yarn-client` Run application in YARN client mode

`spark.task.cpus` Number of virtual cores an executor

Code Listing 5.1: `spark-submit` options used to deploy 28 containers on `ice2`

```
spark-submit \
--master yarn-client \
--conf spark.task.cpus=1 \
--conf spark.yarn.am.memory=4096m \
--conf spark.yarn.am.cores=4 \
--conf spark.yarn.am.memoryOverhead=1024 \
--conf spark.executor.memory=4096m \
--conf spark.yarn.executor.memoryOverhead=1024 \
--num-executors 27 \
--executor-cores 4 \
--conf spark.kryoserializer.buffer.max=256m \
        ⋮
```

	deploys per task
`spark.yarn.am.memory`	Amount of memory to allot the YARN ApplicationMaster
`spark.yarn.am.memoryOverhead`	Amount of off-heap memory for virtual machine (VM) overheads to allot the YARN ApplicationMaster
`spark.yarn.am.cores`	Number of virtual cores to allot the YARN ApplicationMaster
`spark.executor.memory`	Amount of memory to allot each executor
`spark.yarn.executor.memoryOverhead`	Amount of off-heap memory for VM overheads to allot each executor
`num-executors`	Number of executors to launch on the cluster
`executor-cores`	Number of virtual cores to allot each executor
`spark.kryoserializer.buffer.max`	Maximum buffer size for serialized objects used by Kryo

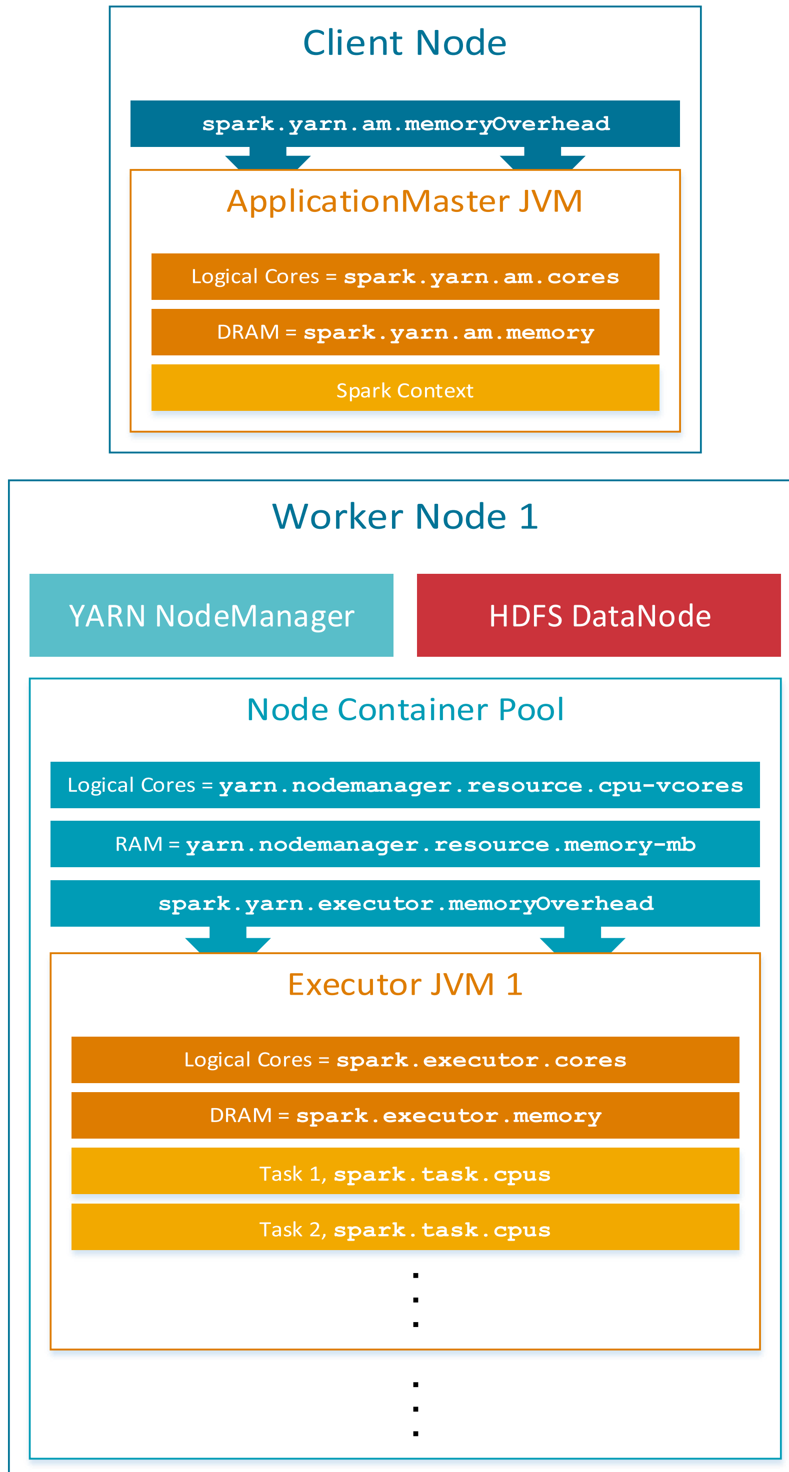

Figure 5.1: Basic Spark configuration architecture. Inspired by [51]

The script shown in Code Listing 5.1 allots each executor a total of 5 GB DRAM and 4 virtual cores, and the aggregate amount of resources allocated is calculated as follows:

The total number of virtual cores C requested is calculated by:

$$C = (\texttt{num-executors} \times \texttt{spark.executor.cores}) + \texttt{spark.yarn.am.cores}$$
$$C = (27 \times 4) + 4 = 112$$

and, given executor memory

$$E_m = \texttt{spark.executor.memory} + \texttt{spark.executor.memoryOverhead}$$

and ApplicationMaster memory

$$AM_m = \texttt{spark.yarn.am.memory} + \texttt{spark.yarn.am.memoryOverhead}$$

the total amount of DRAM $\mathcal{D}$ requested is:

$$\mathcal{D} = (\texttt{num-executors} \times E_m) + AM_m$$
$$\mathcal{D} = (27 \times (4096\ \text{MB} + 1024\ \text{MB})) + (4096\ \text{MB} + 1024\ \text{MB}) = 143360\ \text{MB}$$

5.1.3 Compute-Intensive Workflow

The compute-intensive application is represented by our use case workflow of the previous chapter for analyzing a realistic metagenomic sample dataset. The results serves as a proof-of-concept of COMBUSTI/O facilitating a bioinformatics workflow, and demonstrates the cost of the recovery mechanism at the workflow manager level by running it with the recovery mechanism disabled, enabled, and using it to rebuild the state of a completed workflow. It does not involve other workflows for comparison, but is performed on a realistic dataset to determine wall-clock time consumed, and serves as a proof-of-concept of performing a subset of the META-pipe backend processing using COMBUSTI/O.

The BLAST database splits and InterProScan 5 distributions currently reside in NFS, but could be replicated across all nodes on uniform local paths to diminish networked disk I/O overhead and file contention, but as both tools are compute-intensive it is likely to represent a negligible improvement in performance relative to its total execution time.

Tool Setup

The following paragraphs details the commands, as they are executed using COMBUSTI/O, for the different tools:

Ray The Ray tool is special due to using the external distribution and execution command `mpirun`, and is thus only invoked once at the driver. It accepts `gzip` compressed files as input. As the cluster used is heterogeneous, a `hostfile` containing information about the host and number of CPUs it has available is supplied the command, as well as launching an amount of processes equal to the sum of CPUs on all hosts. Recall that 9 nodes have 8 virtual cores, and 5 have double the amount of that, making the number of processes to launch: $9 \times 8 + 5 \times 16 = 152$. `mpirun` version `1.6.2` (Open MPI) and Ray `2.3.2-devel` was used during evaluation.

Code Listing 5.2: Full Ray command

```
mpirun \
--np 152 \
--hostfile hostfile \
--mca btl_tcp_if_exclude docker0,lo \
Ray \
-k 31 \
-p rayInput/left_input.fastq.gz rayInput/right_input.fastq.gz \
-minimum-contig-length 300 \
-o rayOutput
```

MGA The MGA command is executed using the path to the input FASTA as a command line argument. The version of the 19 August 2008 release was used when evaluating.

Code Listing 5.3: Full MGA command

```
mga_linux_ia64 mgaInput/mga.in
```

BLAST The `blastx` command has some distinct variables that are changed, one of which determines what split to run the BLAST query against, another based on the predicate of what host is running the command. The splits are demarcated by monotonically increasing integers from 1 to 8, and both the database and output names changes for each split; the other, determining the number of threads to launch, is set to either 2 or 4, depending on a `glob` predicate determining what host is running the command, in order to utilize all cores. The version utilized is `2.2.31+`.

BLAST Database The database used for querying is the UniProt UniRef50 protein database, which contains UniRef90 seed sequence clusterings, and is

Code Listing 5.4: Full `blastx` command

```
/usr/bin/blastx \
-db uniref50_split_1 \
-query blastInput/blast.in \
-out blastOutput/blast.out_1 \
-max_target_seqs 5 \
-outfmt 6 \
-evalue 1e-5 \
-dbsize <blast_db_size> \
-num_threads 2
```

a 6.3 GB large uncompressed FASTA file. We downloaded the UniRef50 data 11 March 2016, and the data is publicly available for download on the UniProt FTP server [52]. The source UniRef50 FASTA file was split into 8 smaller FASTA files, using the total number of lines divided by 8 incrementally and choosing the line of the nearest record to determine start and stop offsets. Given the total amount of lines L, and $l \in L$ the line obtained when incrementing by the amount of lines per split ($\sum_{l=0}^{8} l \times \frac{L}{8}$), for each l, the nearest description line h was used to delimit a split to round to the closest record by opting for the minimal line difference in either direction ($|\lceil h \rceil - l| \wedge |\lfloor h \rfloor - l|$), yielding the set of start and stop offsets from line 0 to L, subtracting 1 from the end offsets to avoid overlap except for the last split. All BLAST databases were created using the command as shown in Code Listing 5.5 on each FASTA file split.

Code Listing 5.5: BLAST database creation command

```
makeblastdb \
-in uniref50.fasta_split \
-dbtype 'prot' \
-out uniref50_split
```

InterProScan 5 Similar to `blastx`, the number of cores to utilize is also based on the host on which it runs, and that amount is specified in the configuration file of the InterProScan 5 distribution. Conducive to supporting different numbers of cores for the two host types, the InterProScan 5 distribution was replicated to create two different configuration files residing in two different directories in NFS, directing nodes to the directories with corresponding configurations. We used version `5.14-53.0`.

Code Listing 5.6: Full InterProScan 5 command

```
bash interproscan-5.14-53.0/interproscan.sh \
-goterms \
-iprlookup \
-f tsv\
--applications TIGRFAM,PRODOM,SMART,PROSITEPROFILES,HAMAP,
    SUPERFAMILY,PRINTS,PANTHER,GENE3D,PIRSF,COILS \
```

```
-t n \
-i interProScan5Input/interproscan5.in \
-o interProScan5Output/interproscan5.out \
```

The dataset used for evaluating COMBUSTI/O as the processing engine of our marine metagenomics use case is a marine sediment metagenome sample, "muddy", which represents an anticipated typical dataset of high complexity and medium size [53], having a combined size of approximately 2.6 GB in `gzip` format. Its sample accession is `SAMEA3168559`, the library layout of is paired, and it was generated using the Illumina MiSeq instrument model. The dataset is publicly available for download online from the European Nucleotide Archive (ENA) at EBI's servers (`https://www.ebi.ac.uk/ena/data/view/ERP008945`).

The `spark.task.cpus` variable was increased to 2, in furtherance of launching twice the number of threads per task, thus consuming less memory for both `blastx` and InterProScan 5, and preventing the executor containers being killed by YARN due to memory exhaustion violations. This choice was based on the ability to configure the amount of threads to launch for both `blastx` and InterProScan 5, as the Ray wrapper does not utilize resources as seen by YARN, and the runtime of the single-threaded MGA wrapper is insignificant in comparison. All compute-intensive experiments were run using the `defaultParallelism` multiplied by a factor of 3.

5.1.4 Data-Intensive and Latency-Sensitive Workflows

Both the data-intensive and latency-sensitive evaluations are assayed using three different workflows representing different amounts of optimization; in descending order: a pragmatic Spark implementation and two mirrored functionality workflows implemented in COMBUSTI/O, one wrapping Scala built-ins and another wrapping program binaries. The pragmatic Spark implementation provides best-case running times, and the COMBUSTI/O in-memory implementation wrapping Scala built-ins performs in-memory computations while omitting the disk I/O imposed when using binaries requiring named input and output files, which represents the performance of using `stdin` and `stdout` streams for input and output purposes. These streams may be programmatically manipulated in memory, circumventing disk I/O by writing input to `stdin` upon creating the environment of a subprocess prior to forking, and reading output from `stdout` subsequent of its execution. The other runs of the Scala built-in workflow adhere to the disk I/O pattern imposed by COMBUSTI/O.

We evaluate both the COMBUSTI/O Scala built-in and binary workflows with the recovery on the workflow manager level disabled, enabled, and the recovery

mechanism itself to measure the cost and trade-off of the recovery mechanism for small and large datasets.

The data-intensive workflows demonstrates the throughput of COMBUSTI/O and its I/O pattern on large datasets to ascertain its applicability by calculating observed throughputs, looking into the performance implications of the extensive disk reads and writes, as well as providing a comparison of the subprocess forking and remote tool execution of COMBUSTI/O to the use of Scala built-ins on bigger datasets.

For the latency-sensitive workflows we evaluate its applicability for interactive use, the overhead of recovery for small datasets, as well as the overhead of Scala built-ins and tool execution mechanism of COMBUSTI/O.

To evaluate our framework for data-intensive and latency-sensitive workflows we measure end-to-end wall-clock time consumed by three workflows: our synthetic COMBUSTI/O binary execution example workflow, a pragmatic Spark implementation, and a COMBUSTI/O Scala built-in workflow mirroring the functionality of the pragmatic Spark implementation.

Binary Execution Workflow A parallelized version of our example workflow of Chapter 3 serves as our binary execution workflow. Recall that this workflow wraps the UNIX programs `cat` (version `8.4`), `grep` (version `2.20`), and `wc` (version `8.4`).

Scala Built-In Workflow We also use a workflow that wraps Scala built-in methods in COMBUSTI/O to mimic the different tools, wrapping a method for programmatic writing and reading, to and from files on disk, emulating `cat`; a method filtering on a query (`filter(_.contains("query"))`), imitating `grep`; as well as a method for counting the number of words in the lines producing a hit in the previous stage (`map(_.split(" ").count(_.nonEmpty))`), to simulate `wc -w`. This workflow was designed to closely resemble the different stages of the pragmatic Spark code, with the added disk I/O manipulation and overhead imposed on COMBUSTI/O. The `validateBefore` mechanism is overridden for this workflow, conceptually implemented as a no-op. An illustration of the workflow implementation is provided in Code Listing 5.7.

Pragmatic Spark Implementation The pragmatic Spark way, shown in Code Listing 3.2), does computations in memory without any additional intermediate reading and writing to disk, beyond initially reading the input data and possible shuffling, and hence do not provide any provenance management nor facilitates the use of legacy binary programs, to provide a best-case end-to-end time measurement for comparison.

Data-Intensive Input The data-intensive workflows were run using as input two replicas of the Wikipedia `enwiki` XML dump downloaded 3 May 2016, openly available and hosted on Wikimedia [54], totaling an approximate of 106.4 GB combined size. The pragmatic Spark implementation used the default value of `defaultParallelism` to determine the amount of partitions; the Scala built-in memory-only workflow was evaluated with the variable multiplied by a factor of 30; the rest of the experiments were run using 600 times the default value.

Latency-Sensitive Input To evaluate latencies, we ran the workflows using 12 MB of Web archive (WARC) data as input to the different workflows. The source WARC file was retrieved from the Common Crawl corpus crawl of February 2016 downloaded 3 May 2016, and the data used as input are its first 270,000 lines (`head -n 270000 warc > input`). All runs were performed using the unmodified `defaultParallelism` variable to select partitioning granularity and were run 5 times. The source input file is publicly available on AWS S3:

```
wget https://aws-publicdatasets.s3.amazonaws.com/common-crawl/crawl-data/CC-MAIN-2016-07/segments/1454701145519.33/warc/CC-MAIN-20160205193905-00000-ip-10-236-182-209.ec2.internal.warc.gz
```

Code Listing 5.7: COMBUSTI/O Scala built-in workflow implementation. The code is curtailed for brevity, chiefly omitting details of recovery and parallelization mechanisms, i.e., respectively involving saving an RDD to HDFS and having map-enclosed tool executions like shown in Code Listing 3.7

```
class WorkflowManager(context: => SparkContext) extends Command  {
                                 ⋮

  override def apply(): Unit = {

    val localOutputPath = "/tmp/"
    val jobPath = buildJobPath(localOutputPath, "myJobId")

    // Run ReadWrite on input file
    val readWritePath = buildToolPath("readwrite", jobPath)
    val readWriteInput = ReadWriteInput("file.txt")
    val readWriteContext = new ToolContext {
      def path: String = readWritePath
    }

    val readWriteWrapper = new ToolWrapperImpl(new ReadWrite)
    val readWrite = readWriteWrapper(readWriteContext)
    val readWriteOutput = readWrite(readWriteInput)

    // Run filter on readWriteOutput
    val filterPath = buildToolPath("filter", jobPath)
    val filterInput = FilterInput("query", readWriteOutput)
    val filterContext = new ToolContext {
      def path: String = filterPath
    }

    val filterWrapper = new ToolWrapperImpl(new Filter)
    val filter = filterWrapper(filterContext)
    val filterOutput = filter(filterInput)

    // Run count on filterOutput
    val countPath = buildToolPath("count", jobPath)
    val countInput = CountInput(filterOutput)
    val countContext = new ToolContext {
      def path: String = countPath
    }

    val countWrapper = new ToolWrapperImpl(new Count)
    val count = countWrapper(countContext)
    val countOutput = count(countInput)
                                 ⋮

  }
}
```

5.1.5 Measurements

To evaluate COMBUSTI/O, we measure the performance and overheads of data-intensive, latency-sensitive, and compute-intensive workflows by quantifying end-to-end wall-clock time consumed by running four distinct workflows on different datasets representing the three types of applications, configured with coarse-grained recovery enabled, disabled, and measuring time spent recovering an entire workflow.

We opted for end-to-end measurements over microbenchmarks, because it is hard to correctly do the latter on the Java Virtual Machine (JVM), entailing warming it up with several dry runs, triggering all code to be just-in-time (JIT) compiled and optimized, to circumvent the first couple of runs being slower and straggler-inducing [55], and using some existing framework for benchmarking, e.g., Caliper or Java Microbenchmark Harness (JMH). Moreover, the former provides a holistic illustration of execution times as perceived by the user, because each workflow is to be submitted separately, imposing overheads of starting up Spark and registering containers and launching executor JVMs, including the pre-JIT compiled initial executions that are not subjected to optimizations. Having done tests on a warmed up JVM with JIT compiled and optimized code would produce synthetic best-case results that are unrepresentative of actual usage.

The recovery mechanism at the tool abstraction level for locally assigned output partitions is enabled for all tests except the in-memory Scala built-in workflow, but is not evaluated, and was only tested to work given a hit on the assigned output partition on disk. The end-to-end time is the time consumed by the Spark application for a given workflow, from start to finish, as reported by the YARN Web UI, and is truncated to the nearest second. All evaluation runs deployed the entire cluster, as shown in Code Listing 5.1, unless otherwise stated. Additionally, all results except for the compute-intensive workflows – due to their longevity – with recovery enabled and disabled were run 5 times to acquire more precise measurements.

5.2 Results and Discussion

This section covers the results, discussions, and conclusions of the benchmarks of our evaluational workflows and their different configurations regarding I/O and recovery. We use the workflows denoted in the former to ascertain the performance characteristics of COMBUSTI/O and to get insights in its overheads of I/O management, subprocesses forking for binary execution, and recovery mechanism at the workflow manager level using the results of our compute-

intensive, data-intensive, and latency-sensitive evaluations. Additionally, we investigated cluster-wide disk and CPU utilizations of a data-intensive intensive task to determine if disk I/O is likely to be a bottleneck for our compute-intensive use case.

5.2.1 Compute-Intensive Pipeline Evaluation

To evaluate the utility of COMBUSTI/O facilitating a compute-intensive pipeline, we run our implementation of the marine metagenomics workflow on a marine metagenomic dataset typical for marine bioprospecting. We measure the end-to-end wall-clock running times of this workflow having the recovery mechanism of the workflow manager enabled, disabled, and using the recovery mechanism for state recovery. These experiments are performed to prove the applicability of COMBUSTI/O in compute-intensive bioinformatics workflows and to assess the overhead of the recovery mechanism for compute-intensive workflows, as well as the recovery time itself, to get an exemplar for trade-off reasoning purposes. Note that the running times of having recovery enabled and disabled were only measured once, opposed to the 5 of state recovery.

The results of our compute-intensive experiments are shown in Figure 5.2. There was an issue when running the experiment with recovery off, in which a disk of a single node crashed, having completed 320⁄324 tasks of the workflow, with the remaining four being assigned to the crashed node. Due to time constraints, we were not able to perform another experiment, and as we did not have straggler speculation enabled, the driver was stuck waiting for a response. However, as the tool wrappers are run independently of one another, and the disk crash did not occur on the driver node, we calculated a conservative estimate of the overall running time by adding the maximum running time of a task to the time when the last task was submitted to the faulty node and chose the largest value of this and the latest recorded task completion, based on information logged in the Spark History Server. It is important to emphasize that this is an approximate truth as the workflow was performed at slightly reduced capacity, with the node in question completing about half the average amount of tasks completed on the remaining nodes, likely making the running time larger than it would have been given all healthy nodes.

With the reasoning above, the 1.75% slower running time of having recovery disabled as opposed to enabled is probably not an anomalous result, but rather a product of the reduced resources. Assuming a uniform environment, this would be an estimated loss of 13⁄14 (nodes) for approximately 50% of the duration of the workflow (having completed half the tasks of the average node). Direct comparison of these running times is therefore not precise, but leads us to assume that the overhead of having recovery enabled cannot be very large,

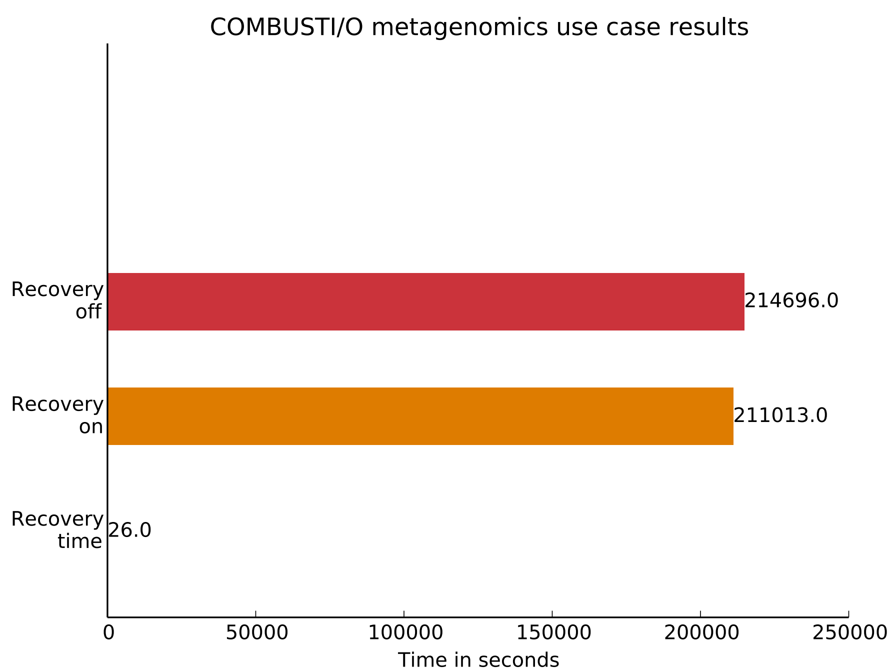

Figure 5.2: Results of the different compute-intensive configurations. The recovery off and on bars represent the end-to-end time of a single run each, while the recovery bar signify the mean of 5 sample runs. Due to the long running times of these workflows, 5 samples could not be acquired. However, we assume the variance of this workflow to be relatively low, based on previous evaluations of META-pipe 1.0 [53]

because the disparity in running times is still relatively small.

In any case, the observed time required to recover an entire workflow of less than half a minute is very fast in comparison to the longevity of the running times of having either recovery enabled or disabled, being almost 4 orders of magnitude smaller, making the recovery option extremely appealing.

There are several possible optimizations to the workflow of our use case, the most pertinent of which is manual translation of nucleotide sequences to protein sequences, because not translating from nucleotides to peptides incurs a lot of additional computation. By virtue of knowing the nucleotide sequence to be the product of gene prediction, there is a 1:1 translation ratio of nucleotides to proteins because we know that the translation is located at frame 0 in forward direction. However, the annotation programs in use are not aware of this detail, and hence the translation is expected to sextuply expand the data, as the translation ratio then becomes 1:6 of nucleotides to proteins. In theory, this

should entail that our measured stage running times being larger than those of META-pipe 1.0 by a factor of approximately 6 (if execution time increases linearly in response to input size) for `blastx` and InterProScan 5, given the same setup. Adding this and support for the Phobius tool of the InterProScan 5 suite, we could begin doing direct comparisons between COMBUSTI/O and the backend of META-pipe 1.0, to further investigate performance implications and characteristics of using Spark and the big data frameworks versus the script-based approach. This is, however, left as future work.

In conclusion, as the time spent computing dwarfs the time consumed performing I/O operations, we argue that the overhead of the I/O pattern enforced by COMBUSTI/O is negligible relative to the CPU time. Moreover, because of the initial dataset being smaller than those used in our tuning experiments (§ 5.2.4), as well as the data reduction of Ray prior to the subsequent stages (reduced to tens of megabytes of contigs), we do not believe I/O performance to be an issue for our marine metagenomics compute-intensive workflow.

5.2.2 Data-Intensive Pipeline Evaluation

Data-intensive pipelines require high throughput, and typically necessitates disk reads and writes, unless the nodes of the cluster upon which the application is run is equipped with abundant amounts of available DRAM. To evaluate the applicability of COMBUSTI/O for data-intensive workflows, we measure the end-to-end wall-clock time of running our binary and Scala built-in workflows on a large dataset, using these results to calculate the respective throughputs having recovery enabled and disabled, as well as measuring the time it takes to recover a completed workflow. We also seek to gain insights in the performance penalty of enforcing the disk I/O pattern of COMBUSTI/O, and the overhead incurred by remote execution using subprocesses in comparison to using Scala built-in functionality on big data. Moreover, to provide a best-case in-memory running time, we also apply the pragmatic Spark pipeline to the same dataset.

The results of our data-intensive evaluation are shown in Figure 5.3, in which throughputs (parenthesized) were calculated based on the average end-to-end running time ($\frac{106.4\text{ GB}}{t}$), and Table 5.4 lists the results including medians and standard deviations. The standard deviation represents fluctuation or variation in a set of measurements, and is the square root of the variance.

The pragmatic Spark, in-memory Scala, and binary execution with recovery off workflows are approximately an order of magnitude apart, in terms of throughput. The in-memory results compares the pragmatic Spark implementation with the Scala built-in workflow in COMBUSTI/O, and as can be seen, the Spark

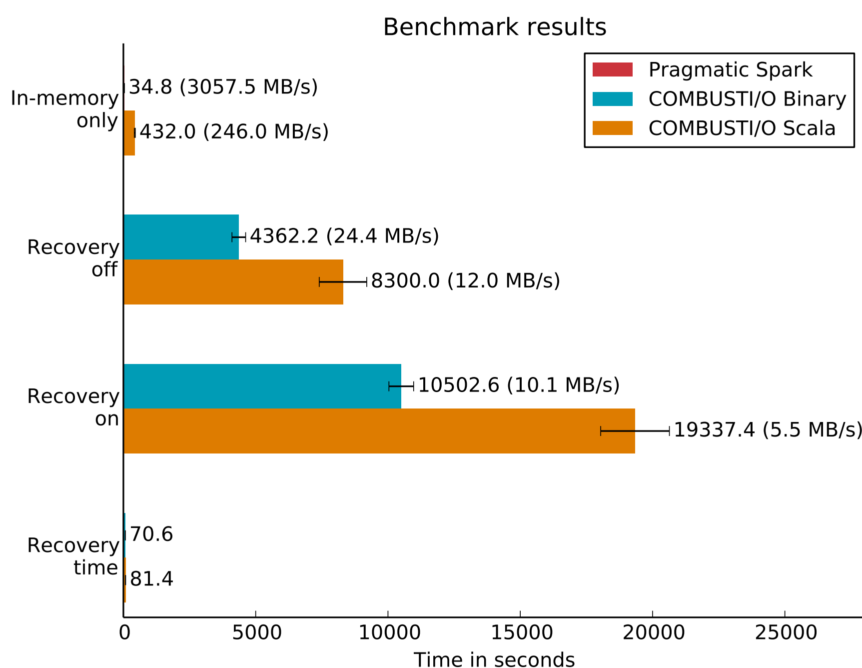

Figure 5.3: Data-intensive results of the workflows using 5 samples. The bars are the arithmetic averages of the samples, and the error bar signifies the standard deviation of the sample set. Note that the uppermost two bars compare the COMBUSTI/O Scala built-in workflow with the pragmatic Spark implementation, and the subsequent three sets of bars juxtapose running times of the binary and Scala built-in workflows of COMBUSTI/O

only implementation outperforms COMBUSTI/O by more than an order of magnitude. The functions of the Spark implementation are likely optimized over several iterations and continuous development of their framework, whereas the ad hoc parallelization of COMBUSTI/O for memory-only computation seems to add a significant amount of overhead in comparison. This is possibly because of the added function call stack of COMBUSTI/O, as well as having the method mimicking `cat` enabled, which reads input into memory and returns it, which is omitted in the Spark only implementation. Moreover, the disk I/O pattern of COMBUSTI/O incurs overhead equal to approximately an order of magnitude, comparing the recovery off binary workflow with the Scala in-memory workflow.

For the recovery off and on configurations, we see through comparison that the COMBUSTI/O binary execution workflow uses half the time of the COMBUSTI/O Scala built-in workflow. This was an unexpected result to us, as we anticipated the built-in functionality to run faster than forking subprocesses

Table 5.4: Data-intensive results of the workflows using 5 samples

Workflow	Mean ($\bar{x}$)	Median ($\mu_{1/2}$)	SD (σ)
	In-memory		
Pragmatic Spark	34.8 s	34.0 s	2.32 s
COMBUSTI/O Scala	432.0 s	430.0 s	18.14 s
	Recovery off		
COMBUSTI/O binary	4362.2	4426.0 s	257.85 s
COMBUSTI/O Scala	8300.0 s	7804.0 s	897.59 s
	Recovery on		
COMBUSTI/O binary	10502.6 s	10291.0 s	467.16 s
COMBUSTI/O Scala	19337.4 s	19284.0 s	1298.83 s
	Recovery time		
COMBUSTI/O binary	70.6 s	69.0 s	2.33 s
COMBUSTI/O Scala	81.4 s	82.0 s	3.88 s

for remote program execution. We therefore speculate that the performance of strings and string comparison on the JVM is poor in comparison to the UNIX tools. Moreover, running with recovery enabled is slower by more than a factor of two for both workflows, suggesting that our recovery mechanism is unsuitable for data-intensive workloads unless the number of failures are likely to exceed 2, which may be the case when testing and debugging. The cost of the recovery is high because it requires triply replicating stage output to HDFS, and is thusly more pronounced for data-intensive applications.

The recovery times measured for both workflows are, as expected, very low in comparison to the original runs. The more than 10% longer recovery times for the Scala built-in workflow can be interpreted as to give some merit to the theory of poor string processing performance in this workflow compared to the binary execution workflow, as the output sizes are similar.

Based on the I/O pattern imposed to facilitate execution of tools requiring named input and output files, we do not expect the throughput to be very high, because it entails excessive amounts of disk writes and reads to and from different locations, effectively yielding random read and write speeds, being further degraded by the redundancy of these operations as performed in

COMBUSTI/O.

COMBUSTI/O suffers greatly from having to do the I/O reads and writes necessary to facilitate tools requiring named input, then writing output to files. It represents a great penalty, but the inefficiency is obvious when the typical I/O manipulation pattern requires input to be read, written, read, written, then read again, for each partition to be processed: Its I/O pattern results in initially reading the input partition from HDFS to memory, then writing it to the node local disk, thereupon the tool is passed the path of the input now residing on disk, which reads it back into memory, performs some computation and then writes its results back to disk, and finally the results are read back into memory to be returned as an output partition.

Somehow consolidating reads and writes to the same locations to amortize I/O overhead through leveraging sequential read and write speeds could be beneficial, but is not straightforward other than increasing the size of each input partition by decreasing the amount of total partitions. A possible optimization is letting the input of a stage to point to the output of a previous, but would require guarantees of data locality, else local paths will not suffice.

Furthermore, random speeds on hard disk drives (HDDs) are impacted by having to wait for read/write heads to move to the correct tracks, and wait for the disk to rotate to the correct sectors. The last point may already in part be counteracted by modern HDDs through extensive buffering and larger buffer sizes of the disk caches, batching operations.

Striping disks (RAID 0) or increasing the number of disks and disk controllers per CPU ratio would likely result in an increase in performance, as a result of COMBUSTI/O's excessive disk usage. Another volatile option for improving performance would be using a DRAM mounted operating system (OS) or to use `tmpfs` for reading and writing, but would require nodes with superfluous amounts of available memory, and would also negate recovery mechanisms. The best alternative is likely reimplementing in data-intensive tools adhering to the principles and schemata of ADAM [20], omitting unnecessary disk I/O as required by COMBUSTI/O to support tools requiring named input and output files by seeking to exclusively perform in-memory computing. COMBUSTI/O is still useful for its purpose, which is to reduce running times of data parallel tasks by facilitating the distributed and parallel execution of unmodified legacy tools and program binaries, supporting the need for named input and output files.

Input data could be piped directly to the tools without any issues (manually or using the Spark-supported `pipe` operation), but most of the tools used in the metagenomic pipeline are not susceptible to this behavior, as they require a

named input file and a named output file, not reading from `stdin` nor writing to `stdout`. Bioinformatics tool writers should therefore be urged to use the standard input, output, and error streams, to diminish the impact of the I/O overhead, allowing for writing to `stdin` and reading from `stdout` conducive to partly circumventing costly disk I/O, thus increasing throughput.

In conclusion, based on the observations, COMBUSTI/O produces an excess amount of I/O overhead for disk reads and writes, making it unsuited for data-intensive applications requiring high throughput. Having the recovery mechanism enabled for large datasets is also questionable, as it more than doubles the running time, and the trade-off of having the recovery to fall back on must be contemplated upon making this decision, as the recovery mechanism is fast per se. The in-memory results of the Scala built-in workflow compared with having recovery enabled corroborates our hypothesis of the overhead generated by the enforced I/O pattern being substantial. Lastly, it is difficult to assess the overhead of the remote execution of tools in the COMBUSTI/O binary workflow when the disparities are as large as observed in our experiments when compared to the Scala built-in workflow, therefore, the takeaway regarding this is that we cannot be certain due to not having thoroughly investigated differences in output sizes nor string performance and optimizations in the JVM.

5.2.3 Latency-Sensitive Pipeline Evaluation

Latency-sensitive applications are typically interactive tools, in which a user interacts with some GUI or Web interface, actively waiting for responses of the tasks being performed, implying that the latency, i.e., the time it takes from a task is launched until the results are returned to the user, must be reasonably low in order to be applicable in practice. An example of such an application is ad hoc exploration of metagenomic data results that may be performed by biologists. We evaluate the utility of COMBUSTI/O for facilitating latency-sensitive workflows by measuring the end-to-end wall-clock time of running the COMBUSTI/O binary and Scala built-in workflows on a small dataset with recovery enabled, disabled, and the time it takes to recover a completed the workflow. Moreover, we want to assess the cost of adhering to the I/O pattern COMBUSTI/O on small datasets, and to establish the overhead incurred of the forking subprocesses for remote tool execution in the COMBUSTI/O binary workflow compared to the Scala built-in workflow on small datasets where this overhead might be more pronounced. To obtain the presumed ideal end-to-end time, we apply the pragmatic Spark implementation to the same dataset.

Spark is shown to be well suited for interactive analysis in the form of repetitive data querying due to only loading the dataset into memory once (if it fits), then

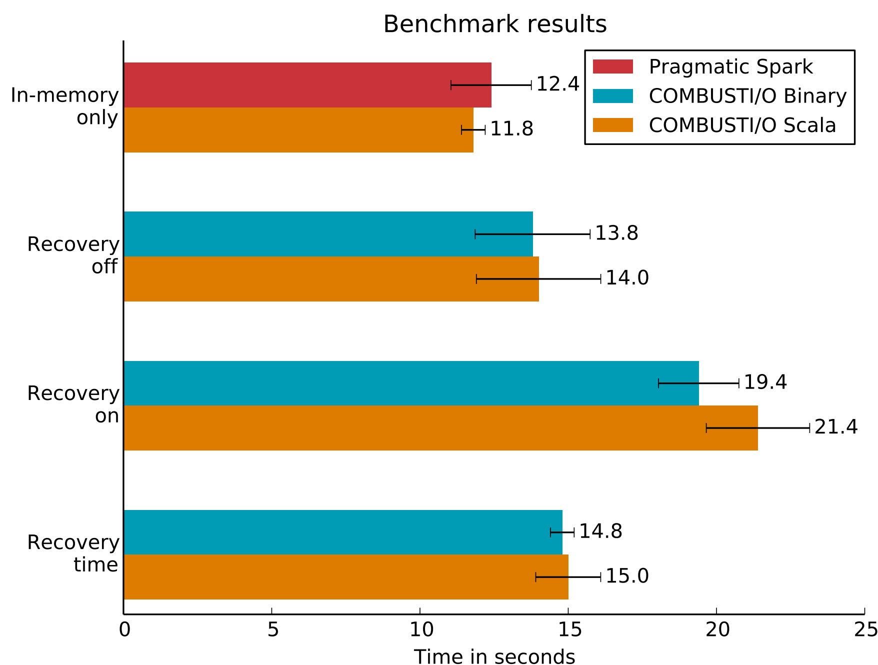

Figure 5.4: Latency-sensitive plot of the different workflows using 5 samples. The bars are the arithmetic averages of the samples, and the error bar signifies the standard deviation of the sample set

being able to query it performing in-memory computing only, which is much faster than having to read from disk the same dataset prior to processing each query [22]. An important distinction to make is that this interactive querying is likely performed using the Spark shell-like REPL, and that our workflows are run using the `spark-submit` tool, which is more likely used for long-running analyses jobs and possibly not that fit for interactive usage.

Hoxmeier et al. [56] found that the response time of an interactive browser-based program is associated with the user-perceived complexity of the task to be performed, stating that response times in the interval of 8 – 12 seconds are acceptable for complicated tasks, and response times exceeding 15 seconds are disruptive to the work done by a user.

The results are shown in Figure 5.4, and the in-memory only experiments show negligible differences in running time between the pragmatic Spark implementation and the in-memory COMBUSTI/O Scala built-in workflow, which omits disk I/O. Both the recovery off and recovery time measurements also show inconsequential difference, inferring that the functionality of forking subprocesses in COMBUSTI/O seems to have little to no impact on performance of

the latency-sensitive measurements.

However, the recovery on measurements show a 3 second discrepancy in mean running time, which is surprising anticipating that the subprocess forking would impose a larger latency than using built-in Scala functions. It may also be attributed the small sample set of only 5 runs, with results possibly converging for more runs, or the way objects are stored to HDFS, as the discrepancy was found with recovery enabled. Moreover, COMBUSTI/O with the recovery mechanism enabled adds a significant overhead over doing in-memory computations only.

Applications in Spark seems to be bounded by the number of nodes employed for computation when tasks are small, and that there is a fixed minimum overhead for Spark to register YARN containers and launch executors. We observed that the actual job completion time as reported by the driver from the terminal typically showed a difference of approximately 10 seconds to the time reported by the YARN Web UI.

A concern using HDFS for small files is the potential skew in data load per node, as the granularity of the Hadoop files are determined by the configured block size. Moreover, for small reads, latency is also of greatly affected by the performance footprint of doing disk I/O, and solid-state drives (SSDs) are far superior in terms of responsiveness to that of the traditional HDDs and also yields greater read and write speeds, both random and sequential, circumventing the latency-inducing mechanic nature of the rotating disks and read/write heads of HDDs.

Based on the observed results, we conclude that having recovery enabled for such small datasets is nonproductive, as the cost of restarting and recovering stage data seems to outgrow the cost of restarting and recomputing everything. Additionally, conforming to the I/O patterns used in COMBUSTI/O produces an overhead of a couple of seconds for this small dataset, and the end-to-end measurements using subprocess forking for remote tool execution does not incur large overheads. Furthermore, for small datasets, the overhead of the COMBUSTI/O framework per se is not large when performing in-memory computations, and both COMBUSTI/O workflows with recovery off are comparable to the best-case running time of the pragmatic Spark implementation. Its marginally disruptive results – from the perspective of the study referred to in the former – yields the ambiguous takeaway that it may be applicable for browser-based interactive applications with recovery disabled.

5.2.4 Spark and I/O Tuning

While performing the evaluation of data-intensive workflows we initially experienced very poor throughput, and found the cluster-wide disk I/O to be very low. We therefore investigated this in furtherance of increasing the throughput by experimenting with input the partition granularity of Spark.

During experimentation on large datasets (§ 5.2.2), aggregate disk I/O increased correlative to increasing number of partitions, which is an unintuitive observation, as batching reads and writes to exploit sequential speeds would seem logical, as opposed to having a myriad of small effectively random reads and writes; to paraphrase, throughput increased parallel to finer task granularity when tested using arbitrary factors in the range of [30, 700] times `defaultParallelism`. Only at a multiplication factor of 700 the Java heap space was exhausted, even though the observed aggregate disk I/O was higher than with 600.

Spark seems to thrive on small tasks, requiring a fine balance of task granularity as having too large tasks leads to poor performance and ultimately having executors killed by YARN, and too small tasks cause garbage collection (GC) churn, leaving no resources for performing progressive work. No network contention was observed when performing these experiments. Graphs of cluster-wide aggregate disk I/O and CPU usage are illustrated in Figure 5.5 and Figure 5.6, and it is worth noting that the CPU utilization is high for a presumed data-intensive task.

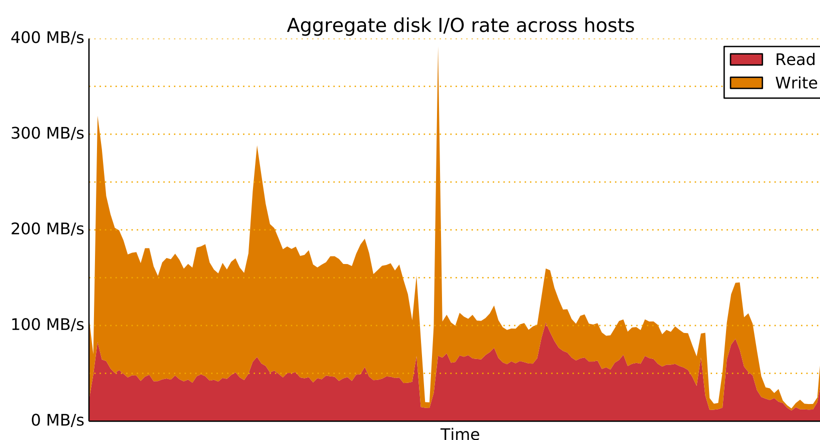

Figure 5.5: Stacked plot of disk I/O of the binary execution workflow from one sample with recovery enabled, as reported in the Cloudera Manager

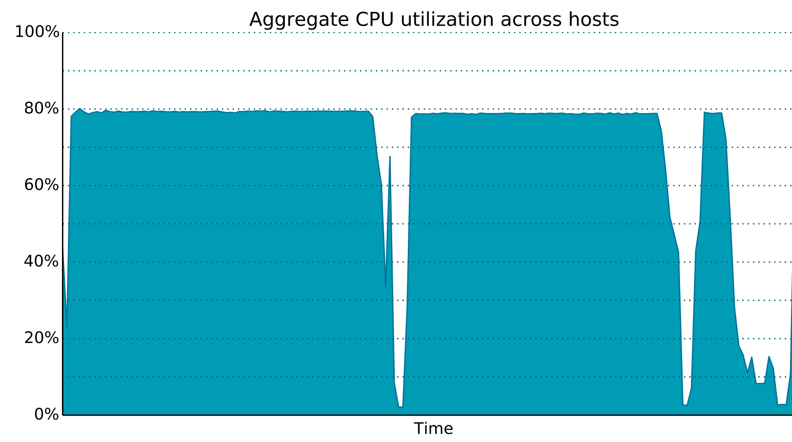

Figure 5.6: CPU utilization of the binary execution workflow from one sample with recovery enabled, as reported in the Cloudera Manager

6

Related Work

This chapter gives elaborate descriptions of the frameworks used in COMBUSTI/O, brief descriptions of basal biological terms and bioinformatics formats, elucidations on the four bioinformatics tools wrapped to implement the marine metagenomics use case, related bioinformatics pipelines and frameworks, information on the ELIXIR platform, and finally a description of the largest metagenomic study to date—*Tara* Oceans.

6.1 Frameworks Utilized

The following subsections provide more detailed descriptions the frameworks utilized by COMBUSTI/O, including Apache Hadoop YARN and HDFS, Apache Spark, and NFS.

6.1.1 Apache Hadoop

Hadoop [23] is an open-source implementation of the proprietary Google MapReduce [57] framework. It is designed to be run on large Linux clusters composed of machines using commodity hardware providing reliability, fault tolerance, and high availability, and is commonly used for distributed data analysis at scale. There are four modules included in Hadoop: Hadoop Common, Hadoop YARN, HDFS, and Hadoop MapReduce.

Yet Another Resource Negotiator

YARN [41] is the resource manager of Hadoop and applications can run on top of and rely on it for scheduling and resource handling in a scalable and efficient manner, while also enabling the sharing and simultaneous use of a cluster by large numbers of frameworks. Its architecture consists of a per cluster ResourceManager that dynamically allocates containers to applications to be run on arbitrary nodes in the cluster, in collaboration with NodeManagers running on the worker nodes, which are responsible for the inhabited node's resources. A container is an abstraction that comprises the delegated logical resources. An ApplicationMaster is responsible for coordinating the execution plan by requesting resources from the ResourceManager and do the execution of a program in a fault tolerant manner, and is itself run as a container in the cluster.

Hadoop Distributed File System

The HDFS [35] is a file system developed to run on a clusters of commodity machines and is designed to support data-intensive workloads in a distributed environment, and is optimized for large files. It is scalable, reliable, highly available, provides accumulative performance, and employs a multiple readers/single-writer model for ensuring file consistency (append and read only). A NameNode, DataNodes, and the HDFS client constitutes the architecture: the NameNode is responsible for maintaining file system metadata and the name space hierarchy; the DataNodes stores the blocks of HDFS files in their local file systems and exposes commands the NameNode can use for organizational purposes; the HDFS client contains code exposing the file system API to applications and handles read and write requests. Identifying that failures are commonplace in large distributed systems emphasizes the need for fault tolerance and recovery and HDFS supports a variety of redundancy mechanisms, including CheckpointNodes, BackupNodes, replication, and block placement strategies.

6.1.2 Apache Spark

Spark [21, 22, 58] is a cluster computing framework written in the Scala programming language, originally built on top of the Mesos [40] platform. It was developed by the AMPLab [59] of UC Berkeley and serves as the processing engine of their Berkeley Data Analysis Stack (BDAS). The BDAS' raison d'être is making sense of big data, and consists of several self-built and third-party components. Spark is designed to support applications unable to be efficiently expressed as acyclic data flows, such as graph processing and machine learn-

ing, in a scalable and fault tolerant manner similar to MapReduce. It does so by introducing the RDD [21] abstraction and parallel operations on these datasets. This abstraction and its accompanying interface allows for efficiently expressing several existing programming models, including MapReduce, SQL, and Pregel [60].

RDDs are read-only (i.e., immutable), partitioned collections of objects created through operations on data in stable storage or other RDDs. This parallel data structure enables data reuse by persisting intermediate results in memory, improving the performance of several types of applications, most notably iterative algorithms and interactive data mining tools [21]. The RDDs expose an interface of *transformations* that executes an operation on many data elements. Transformations define new RDDs and are lazily computed to support pipelining. Efficient fault tolerance is achieved by logging a dataset's lineage instead of the data itself – the lineage consists of all transformations used to build a certain dataset – which allows for recomputing lost partitions, without the overhead of replication. Upon failure, only the missing partitions are recomputed in parallel, if possible. After creation, an RDD can be manipulated using operations that return a value to the driver program or to write data to a storage system, referred to as *actions*. An RDD is represented as a Scala object in Spark, statically typed and parametrized by an element type.

Table 6.1: Examples of transformations in Spark. Derived from [21]

Transformation	Effect
`map` (f: T ⇒ U):	Each item is passed through the provided function `RDD[T] ⇒ RDD[U]`
`flatMap` (f: T ⇒ Seq[U]):	Like `map`, but can map each input to multiple output `RDD[T] ⇒ RDD[U]`
`filter` (f: T ⇒ Bool):	Filter dataset based on the provided function `RDD[T] ⇒ RDD[T]`

In Spark terminology, a developer writes a driver program that connects to a cluster of workers. At runtime, a driver program containing the control flow of an application launches multiple workers that read data from some distributed file system and may persist to memory the computed RDD partitions. The Spark scheduler builds a DAG consisting of execution stages based on the lineage graph of the RDD upon which the action is performed [21]. Tasks may also be scheduled based on data locality using delay scheduling, and in case of memory exhaustion, data is spilled to disk and the performance of the RDDs gracefully degrade [21]. Spark was designed to enhance the Hadoop stack

Table 6.2: Examples of actions in Spark. Derived from [21]

Action	Effect
`count` ():	Count elements in dataset `RDD[T] ⇒ Long`
`reduce` (f: (T,T) ⇒ T):	Aggregate elements of dataset using a provided function `RDD[T] ⇒ T`
`save*` (*path*):	Save dataset to storage system `RDD[T] ⇒ storage(`*path*`)`

and thus supports HDFS, HBase [61], SequenceFiles, as well as AWS S3 [36], amongst others.

RDDs can be persisted in memory either as serialized data or as deserialized Java objects, and it can be persisted in non-volatile storage. Furthermore memory is managed using the least recently used (LRU) eviction policy on RDDs currently residing in memory. By storing Java objects in memory, the cost of deserialization and I/O can be circumvented, making Spark perform better than Hadoop MapReduce in graph and iterative machine learning applications [21].

Spark also includes access to, and interfaces for, GraphX [62], MLlib [63], and Spark Streaming [64], to mention some.

6.1.3 Network File System

The NFS [39] is a distributed file system, originally developed at Sun Microsystems, which was designed to be portable, opaque to the end-user, support recovery following events of failure, and to maintain reasonable performance. The file system components and protocol allows for transparent access to remote file systems using the External Data Representation (XDR) specification to define machine independent protocols and integrates remote procedure calls (RPCs) for remote interaction. NFS consists of interfaces for the Virtual File System (VFS) and for vnodes, both implemented and added to the kernel in order to achieve the user-perceived transparency, as well as a protocol, a server side, and a client side.

The file system has been through several iterations of improvements, and the current latest version is version 4 (NFSv4), defined in RFC 3530.

6.2 Bioinformatics Pipelines and Frameworks

In this section we describe some of the most relevant and popular bioinformatics pipelines for doing metagenomics analyses, which are related to the work conducted in this thesis.

6.2.1 META-pipe 1.0

META-pipe 1.0 [53] is an automated pipeline developed at SfB (UiT) for annotation and analysis of metagenomic and genomic sequence data, primarily a domain-specific service in that it targets marine metagenomics. It has integrations for Galaxy [65], Metagenomics Reports (METAREP) [66], the Stallo supercomputer, and the Norwegian e-Infrastructure for Life Sciences (NeLS) [67]. Galaxy and Taverna [68] are two of the most popular bioinformatics workbenches [25], and Galaxy was chosen over Taverna largely due to the former being the most familiar workflow manager among the collaborators. It also provides a GUI which circumvents the need for programming and CLI manipulation. The design philosophy of META-pipe 1.0 includes utilization of existing frameworks and infrastructure services whenever feasible, and ease of both development and deployment, and the main motivation for developing the system was to efficiently produce full length annotated genes from metagenomic assemblies, as there existed no alternatives that offered the desired properties of flexibility, extensive annotation and visualization options. META-pipe 1.0 currently supports analysis of whole genome shotgun sequence and 16S amplicon data, both taxonomic and functional. The pipeline itself consists of modules for pre-processing, assembly, taxonomic classification, and functional analysis, all of which are modular components that can be configured and combined to the individual user's preference. Figure 6.1 depicts an overview of the main components and modules of META-pipe 1.0.

The default first stage is pre-processing, in which the system is given as input raw sequencing data to be subjected to data cleaning and quality control. The following stage is taxonomic classification, producing an overview of present organisms in the input dataset, succeeded by assembly, constructing a set of contigs (from contiguous, "the result of joining an overlapping collection of sequences or clones" [69]) for functional analysis. Subsequently, functional analysis uses the contigs produced to predict a set of genes and conjoining the genes with the corresponding functional annotations. The final stage is visualization of the produced output, using a Web-based interface for viewing, browsing and various data set manipulation methods.

META-pipe 1.0 is openly available for students and academic employees with a Feide [70] account at `https://galaxy-uit.bioinfo.no/`.

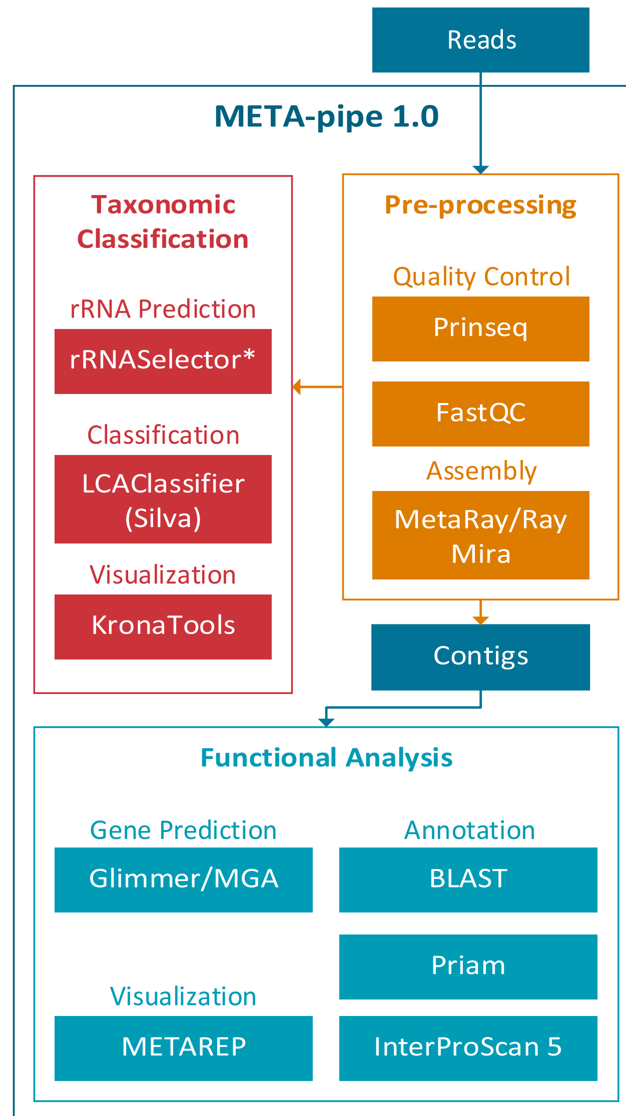

Figure 6.1: Schematic overview of META-pipe 1.0, including components and major modules. Adaption from [53]

The prompt for further development and improving upon version 1.0 was predominantly galvanized by its manual failure handling, having built from scratch a distributed runtime framework for data management and parallel job execution using the Perl and Bash scripting languages, and its lack of provenance management. This, addressing portability concerns, and enhancing the UI, forms the basis for the new META-pipe, which architecture and design is built around rectifying said deficiencies.

6.2.2 ADAM

The ADAM [20, 26] framework is developed as a collaborative effort between UC Berkeley, MIT, and Harvard, originally designed to take on big data genomics, and is made open-source on their Web sites [71]. Their latest work include revising data-intensive analytics as a whole, providing case studies in astronomy and genomics, and more broadly discussing the current trends and road ahead for scientific data analysis and processing systems, introducing a stack model for scientific computing, sanctioning data independence, computation being pushed to data, and quick parallel metadata access.

Within bioinformatics, specifically genomics, they argue the inefficiency of the current de facto standard genomics formats for distributed computing, attributing it to their being initially designed for sequential single-machine processing. Having non-splittable formats does not scale well to the size of current big data, hence ADAM introduces novel, backwards compatible, formats, patterns, and schemata for big data genomics processing designed for distributed and parallel computing at cloud scale using horizontally scalable methods without sacrificing result accuracy.

ADAM is built on top of the Spark big data processing engine, using the Avro serialization system for data representation within the Parquet columnar storage format. Avro provides fast serialization in binary data formats, human-readable schemata, and compatibility with several programming languages; Parquet supports predicate pushdown, high compression, facilitation of parallelism through independent reads as row data is written to disk in fixed size parts, and data may be queried directly using for example Spark SQL. They seek to separate data formats and parsing from processing and representation using their schema for dividing concerns, and achieves speedups in processing using their formats largely because they are able to do iterative in-memory processing of their pipelined stages, omitting the need of having to write and read data to and from disk in between each stage, making the total time spent writing amortized over the entire execution.

Using ADAM would entail reimplementing all tools to be used for a given workflow in Scala, as well as adhering to their formats, schemata, and separation of concerns. One of the principal design goals of COMBUSTI/O is using existing and unmodified tools and frameworks wherever applicable, in favor of not having to reimplement and maintain locally implemented and optimized custom programs, but rather having the goal of being able to run any binary provided by a scientist.

However, ADAM does not incur the penalizing disk I/O to the extent of COMBUSTI/O, as it does not require writing data to disk, before reading it back

in to memory, prior to processing it. Therefore, and because its tools are written in Scala for Spark using natives, splittable formats, and Spark built-ins rather than doing ad hoc parallelization, the performance and throughput of the ADAM framework is doubtlessly better than COMBUSTI/O, as it can perform its execution mostly in memory. Whether or not ADAM plan on expanding their framework to include tools for analyses in the field of metagenomics is uncertain, but it would be of great interest for ELIXIR to get involved with this work and possibly establish a collaboration if this scenario became a reality.

6.2.3 CloudBurst

CloudBurst [18] is an open-source parallel read-mapping seed-and-extend style algorithm, developed at the University of Maryland, for genotyping, personal genomics, and single-nucleotide polymorphism (SNP) discovery, optimized for NGS data. It is modelled after the RMAP program for short-read mapping and utilizes Hadoop MapReduce and HDFS for execution parallelization and data management across nodes, exhibiting linear scalability in running times over RMAP, improving performance by a factor of more than a hundred. Taking as input a multi-FASTA file containing reads and a multi-FASTA file containing reference sequences, its map phase consists of emitting tuples of *k*-mers and reference sequences, and the reduce phase creates shared seeds to end-to-end alignments through extension.

Their evaluation also showed similar speedups doing ad hoc parallelization using the RMAP program on input splits, as compared to CloudBurst, and their implementation is available as a Hadoop MapReduce algorithm parallelization model.

CloudBurst is implemented in Hadoop MapReduce, which is a major limiting factor in terms of its supported execution flow being restricted to map and reduce stages, as compared to Spark which is used in COMBUSTI/O. Moreover, their implementation only cover a read-mapping algorithm, which is only analogous to a single stage in our metagenomics pipeline, but made available their approach and parallelization model for adding other tools, but like ADAM it also would require the reimplementation of the remaining tools required for a given workflow.

6.2.4 Crossbow

Crossbow [19] is a software tool implemented in Hadoop using its streaming module and HDFS for creating a seamless automatic pipeline combining modifications of the already existing Burrows-Wheeler transform (BWT) aligner

Bowtie and the SNP caller SOAPsnp software to be run in parallel on split input across multiple nodes using standard input and output streams, primarily limiting their scope to human resequencing and SNP detection. Bowtie provides fast mammalian genome alignment of short reads with conservative memory usage; SOAPsnp uses aligned short-reads to provide calls of haploid and diploid consensus. It takes as input a preprocessed file wherein reads are tab-demarcated tuples, the map phase consists of alignment using Bowtie, and the reduce phase consists of SNP calling using SOAPsnp, resulting in a stream of tuples containing SNPs.

Moreover, their tests showed Crossbow achieving high precision and sensitivity for alignment and SNP calling, and a high agreement level when compared to the results of the SOAP and SOAPsnp combination.

Similar to COMBUSTI/O, Crossbow also performs ad hoc parallelization of programs by splitting input data and performing the execution of programs on each split, but like CloudBurst it uses the limiting Hadoop MapReduce framework. Additionally, their ad hoc parallelization does not support the use of unmodified program binaries, rather opting for customizing the tools they used for their workflow, meaning that to create our metagenomics pipeline, customized versions of all tools to be used would need to be implemented.

6.3 Biology and Bioinformatics Glossary

The ensuing paragraphs contain explanations of general concepts and terms in biology and bioinformatics.

DNA Deoxyribonucleic acid is one of two types of nucleic acids, which are the molecules holding genetic information, as well as being responsible for its expression, transmission, and storage. DNA molecules are composed of sequences of repeating nucleotides that constitutes the genetic information. Each nucleotide consists of a sugar molecule, a phosphate molecule, and a nucleobase, and the nucleotides present in DNA are the two purine bases adenine (A) and guanine (G) as well as the two pyrimidine bases cytosine (C) and thymine (T). A DNA molecule consists of two nucleotide chains which are held together by hydrogen bonds of G-C and A-T base-pairs, forming the widely identifiable double helix structure [72]. A gene is a DNA nucleotide sequence specifying one polypeptide chain's ordering of amino acids, and a genome is the DNA's encoded genetic information in its entirety. DNA sequencing is the deciphering of an organism's genetic information by determining the correct ordering of the nucleotides. With the next-generation sequencing technologies (non-Sanger

methods), efficiently performing low-cost sequencing of DNA (whole genomes) helps drive several research fields, including biology and medicine [73].

RNA Ribonucleic acid (RNA) is the second type of nucleic acids, and differs from DNA in that it only consists of one nucleotide chain and that the nucleotide thymine (T) is replaced with uracil (U) which may form A-U base-pairs, making the nucleotides present in RNA adenine (A), guanine (G), cytosine (C), and uracil (U). In addition, the sugar molecule of the nucleotides in RNA is ribose instead of DNA's deoxyribose [72]. The three main classes of RNA are messenger RNA (mRNA), ribosomal RNA (rRNA), and transfer RNA (tRNA).

Metagenomics A genome contains all of an organism's genes; its complete set of DNA. Genomics is the analysis of genomes on the organism-level, in furtherance of better understanding the means of the organism and its evolution, but is argued to be insufficient because only a small fraction of microbes can be cultured [74]. Metagenomics seek to understand complex communities of organisms by analyzing their genetic composition, thus capturing the dynamics of these communities in a way genomics cannot, and is also believed to help clarify the questions "Who is there?" and "What are they doing?" [74], which may contribute to the fields of life and earth sciences, and biomedicine, amongst others.

FASTA FASTA is a file format used for storing DNA sequencing data, which consists of records with a description followed by a sequence of nucleotides. A FASTA sequence consists of two fields: (1) a description line that is distinguished by always beginning with the ">" (ASCII 0x3E) symbol, (2) sequence data of arbitrary length [75]. Blank lines are disallowed and sequences should be represented using IUPAC single letter codes. A FASTA file contains one or more sequences and are represented as plain text.

FASTQ FASTQ is the de facto standard file format for storing and sharing DNA sequencing data [76]. It improves on the FASTA format by including quality scores from the Phred quality scale to each nucleobase in all sequences. One read in the FASTQ format consists of four lines: (1) title and optional description, (2) raw sequence, (3) optional repeat of first line, (4) quality scores. The sequence here is also recommended to be represented using IUPAC codes and, like FASTA files, FASTQ files are plain text.

***k*-mer** In nucleotide sequence analysis, k-mer analysis can be used for sequencing coverage estimation, repeat detection, and preparation for de novo assembly [77]. It is defined as all substrings of a DNA read sequence of length k, and is a specialization of the n-gram concept of computational linguistics. An example is given below in Code Listing 6.1.

Code Listing 6.1: Example *k*-mers

```
// Original string
ACAGTCA

// K-mers, k = 4
ACAG
 CAGT
  AGTC
   GTCA
```

However, most contemporary assembly algorithms represent *k*-mer prefixes and suffixes as de Bruijn graphs, as representing all *k*-mers in full does not scale to the very large [78].

6.4 Bioinformatics Tools Wrapped

The following describes the four bioinformatics tools wrapped – Ray, MGA, BLAST, and InterProScan 5 – in order to implement a workflow reflecting the marine metagenomics use case evaluated.

6.4.1 Ray

Ray [79] is an open-source assembler that, simultaneously, assembles short-reads in parallel from a mix of sequencing technologies. Sequencers generate short-reads by decoding fragmented DNA, and assemblers assemble said short-reads into longer sequences of contiguous genomic regions, which is a requirement to obtain a genome sequence of high quality. Accurately assembled genomes are often prerequisites for further analysis. Ray can perform the assembly of single-end, paired-end, and interleaved paired-end reads. Paired reads contains knowledge of the DNA strand of each read and of the sequence content of the reads in pair, and pairs of short reads have previously been deemed satisfactory for de novo sequencing [79].

Ray is a de Bruijn assembler, in the sense that it uses a de Bruijn graph for representing sequences, motivated by the fact that there occurs lots of overlaps between reads, and that this type of graph is memory efficient (dictated by the length of the genome) in the presence of redundant information. The vertices of a de Bruijn graph consists of a *k*-mer, the arcs consists of $(k+1)$-mers, and the value of k is fixed on a per-run-basis and may be specified as an input parameter, the value of which – according to the authors of Ray – should not be a smaller value than 19 to remain sensible. It also supports hybrid assembly, which is

a term used to describe the usage of several different sequencing technologies simultaneously, favorable to diminishing the amount of correlated errors in the reads. The assembler uses several automatically calculated parameters to increase popularity and broaden the user base, including local minimum, local maximum, average fragment library length, and the tolerable difference between the length of a particular pair of reads and the average length. Ray utilizes MPI for scaling purposes, in order to efficiently handle the large amounts of data produced by contemporary sequencers. The metagenomic extension of Ray is called Ray Méta, and is embedded in the original program.

6.4.2 MetaGeneAnnotator

MGA [80] is an ab initio gene prediction tool based on the hidden Markov model that precisely predicts genes in anonmyous phage and prokaryotic genomes by detecting patterns of ribosomal binding sites (RBSs) that are species-specific. It is operated from the command line, and can predict prokaryotic genes from one or more various length genomic sequences, includes statistical models of prophage genes to sensitively detect both typical and atypical genes, and provides a novel RBS analysis approach that enables precise and sensitive prediction of translation starts of genes while maintaining high specificities. The output reports information on RBSs as well as on gene locations used in both metagenomic analyses and for annotation of phage and prokaryotic genomes.

MGA is a further development of MetaGene [81], a metagenomic tool that uses di-codon frequencies and guanine-cytosine content to create logistic regression models for gene prediction using prokaryotic genome sequences as input, and improves on it by adding prophage gene models and an adaptable RBS model that allows for precise prediction even on short and anonymous genomic sequences.

6.4.3 Basic Local Alignment Search Tool

BLAST [82] is a novel sequence comparison approach that, based on optimizing the maximal segment pair (MSP) score, directly approximates alignments as they would be produced by a dynamic programming algorithm based on the same measure. The similarity score is the sum of similarity values for each aligned residue pair of two aligned segments, which are contiguous sequences; the MSP is the pair with the highest similarity score of two aligned segments of same length; and the MSP score quantifies local similarity of any sequence pair, and is sought heuristically computed by BLAST. Local similarity measures are typically preferred over global similarity measures when used for searching

databases [82].

BLAST is chiefly used for various sequence similarity searching schemes, including searches in databases for DNA and protein sequences, motifs, and gene identification, but can also be used for analyzing similarity regions in DNA sequences. It includes optimizations for searching databases more efficiently by reducing time spent on regions of sequences whose similarity to the query are unlikely to surpass the highest MSP score. The running time of BLAST is the cumulative time it takes to create a list of words that, when compared to the words in the query, are presumed able to exceed a score higher than the given threshold, to find word matches from said list in the database, and ultimately to identify the set of matches that have segment pairs with a score higher than the cutoff.

6.4.4 InterProScan 5

InterProScan 5 [83] is a CLI tool for protein function classification at the scale of genomes by enabling the use of multiprocessor machines and clusters for distributed and scalable data analysis. It is used for obtaining potential functions of protein sequences through summarizing the results of a variety of search applications that it incorporates. Search applications are generally divided into two subcategories, based on their approach: those that perform a single algorithm for specific feature prediction, and those that uses several algorithms for multiple model search per sequence also requiring some post processing; InterProScan 5 includes both.

InterProScan 5 also enables the parallelization of search jobs along three dimensions to efficiently characterize and analyze modest amounts of sequences using a spoke-hub distribution model, in which a master communicates with some number of workers, obtaining scalability through preventing congestion by – in addition to the master – allowing workers to spawn new layers workers, allowing for an arbitrary amount of layers. The parallelization is conducted through repartitioning sequence sets for finer granularity to increase parallelism, running different analyses in parallel, and enabling applications that natively exploit parallel computing. The tool is developed and written in Java, utilizing Apache ActiveMQ JMS for IPC purposes, Hibernate for the relational database schema, and JAXB for the XML schema, only leaving one requirement in that the nodes have to share the same file system.

6.5 *Tara* Oceans

Various world ocean ecosystems were sampled during the four year-long scientific voyage of the *Tara* Oceans [84] Expedition (2009–2013), and in total more than 35,000 samples of seawater and plankton and 13,000 contextual measures were collected from the surface of the ocean and the mesopelagic zone, making it the largest modern-day plankton collection of its kind and the largest metagenomics dataset of today. The samples contain millions of small organisms, sampled in 20 biogeographic provinces covering 210 ecosystems, and the raw and validated data sets are promptly made open access [85]. The raw sequencing data is hosted by the ENA, which is the EBI's short read archive, and fast release policy of this data is intended to promote sharing and collaboration, to maximize discoverability, and to expedite exploration, facilitating holistic analyses.

One of the incentives for doing sampling of sea water and plankton on a global scale is the fact that life originated in the sea – which at least is the most widely supported explanation of evolution within science and academia – meaning the ocean contains large quantities of untapped evolutionary knowledge. Plankton is arguably as important as the rainforest for the climate on Earth, and yet so little is known about how these tiny organisms, that cooperate in large numbers converting CO_2 to O_2 in bulk, actually function. This in addition to the potential of analysis of eukaryotes, prokaryotes, and viruses of the oceans have already been demonstrated by a series of publications [85]. The latent information within these datasets are believed to be of relevance to several fields, including bioenergy, nanotechnology, nutrition, and pharmaceutics, amongst others [84].

We plan on using the *Tara* Oceans data as input to the META-pipe pipeline for functional analysis purposes contributing to populating the MarCat marine reference database.

6.6 ELIXIR

ELIXIR is an infrastructure platform constituting the EMBL-EBI in conjunction with various national bioinformatics institutions, having joined forces to increase life science research capabilities. ELIXIR's work primarily consists of forming and coordinating a coherent infrastructure encompassing a collaborative effort between Europe's national and international research resources for analyses and archiving of big biological heterogeneous data [86]. Their open-access infrastructure provides easy access to sustainable, community-standard data resources, and is intended to play an important role in life science projects

across Europe for research purposes within medicine, bioindustries, and society. It incorporates the EMBL-EBI – under which ELIXIR is a special project – and several Nodes, the notion of which are national bionformatics centres including projects from the institutions' affiliated life-science communities.

ELIXIR.NO [87] is a project launched conducive to developing a Norwegian bioinformatics infrastructure to serve as the Norwegian ELIXIR Node, and is financed by the Research Council of Norway (RCN). The project officially launched 1st October 2012, organized by the University of Bergen in cooperation with the Universities of Tromsø, Trondheim, Oslo, and Ås. In addition to being backed by the RCN, it is also partially funded by the partaking academic institutions, and is a further development of a bioinformatics technology platform which was the product of a partnership between the Universities of Bergen, Tromsø, Trondheim, and Oslo spanning the decade antecedent to the project start. In addition to inaugurating a national ELIXIR Node, the goals of the project includes continuation of the Help Desk service and providing an infrastructure delivering genomic scale data analysis, storage, and publishing services.

Among the obligations of the Norwegian ELIXIR Node is marine metagenomics, assigned the SfB at the University of Tromsø, which involves marine metagenomics research and bioinformatics services. SfB is a cross-disciplinary biotechnology and computer science service center at the Faculty of Science and Technology, comprised of researchers and students from two research groups: Molecular Biosystems Research Group (MBRG) [88] affiliated with the Department of Chemistry and Biological Data Processing Systems (BDPS) [89] affiliated with the Department of Computer Science. The focal point of the research of the latter within SfB is experimental creation and evaluation of metagenomics analysis pipelines beneficial to developing data processing systems of scalable, flexible, and interactive nature.

Conclusion

This thesis presents COMBUSTI/O, a general framework for distributed workflow generation and ad hoc parallelization of serial program binary execution, constituting abstractions that facilitate parallel execution of programs implementing common I/O patterns in a pipelined fashion as workflows in Spark, chiefly supporting compute-intensive applications.

We have elaborated on the motivation for doing this work, which is largely due to the explosive growth in biological data generation and the comparably lower downstream analysis capabilities, and identifying that several bioinformatics tools in current use are sequential programs that – if data parallel – may be distributed across several nodes and executed in parallel in order to diminish wall-clock execution time of analysis pipelines. This entails splitting input data, disseminating the resulting partitions across participating nodes of a cluster, and performing parallel execution of specified tools on each input data partition in a fault tolerant and horizontally scalable manner.

In our experience operating META-pipe 1.0, we have identified some issues that needed addressing in the next iteration of our big biological data analysis service. One of these issues is the nuisance of having to recompute stages following failure, and is commonly encountered for long-running workflows and during testing, debugging and developing workflows. Furthermore, workflow failures ofttimes involve the tools themselves, and may be mitigated by inspecting their correct installation, including dependencies, prior to running a workflow. Additionally, to support a wide variety of tools, flexibility in abstrac-

tion interfaces and I/O handling is of crucial importance, also helping ease the addition of new tools.

We have described the COMBUSTI/O architecture and its place as the cluster scheduler in the new META-pipe architecture, wherein COMBUSTI/O is currently being used as part of an effort migrating the entire big biological data analysis backend to run solely on Spark.

The design of COMBUSTI/O was detailed showing the interaction between the internal components using examples of a workflow, descriptions of the tool abstraction and tool wrapper modules, in addition to code excerpts for clarification of concepts and execution logic. We also detail the metagenomics use case for which the framework is originally designed, inclusive of the tools wrapped for creating the workflow mimicking a subset of the original META-pipe processing backend, the command line arguments altered, and the workflow itself. It is also worth noting that the flexibility of COMBUSTI/O extends to combinations and hybrid solutions of different tools, as the tool wrappers are independent and linked together in a workflow manager. There are no limitations to what may or may not be combined, including but not limited to workflows involving combinations of Spark, Scala and Java built-ins or programs, and different program binaries.

We evaluated COMBUSTI/O using synthetic workflows with wrappers written for UNIX binaries and Scala methods to ascertain how COMBUSTI/O performed for data-intensive and latency-sensitive applications, showing that its throughput is modest for data-intensive tasks due to its imposed I/O pattern of disk reads and writes, and that it incurs latency defined as disruptive to interactive work for latency-sensitive tasks. The evaluation of its applicability in compute-intensive applications were performed with the marine metagenomics use case, demonstrating that COMBUSTI/O is flexible and can support a wide range of I/O patterns, and that the coarse-grained recovery mechanism is also well-suited for compute-intensive applications, and applications wherein several failures or crashes are expected, like development, debugging and testing.

By building on top of established frameworks supporting rich sets of operators for big data analysis and management, COMBUSTI/O leverages the mechanisms of these for fault-tolerance and scalability, as the abstractions and workflow manager of COMBUSTI/O are run independently of one another, making these characteristics contingent on the underlying frameworks. COMBUSTI/O also supports recovery mechanisms on task and stage granularities, analogous to results of tasks being persisted locally and stages on HDFS, to facilitate restarting workflows following events of failure. Additionally, COMBUSTI/O performs validations on all involved tools prior to running a workflow,

to detect tools that are not installed or incorrectly configured, and missing dependencies. Also, our abstractions are flexible enough to support easy addition of tools and complex I/O patterns, including tools requiring named input and output files, and facilitation of standard stream redirection and handling.

To conclude, COMBUSTI/O is a scalable general framework appropriate for representing and realizing complex compute-intensive, data-intensive, and latency-sensitive applications that supports tool validation, two-level recovery, and flexible I/O handling. It allows for easy addition and parallelization of unmodified tools with common I/O patterns and workflow creation using said tools by splitting input data into smaller pieces, evenly distributing the pieces among participating hosts, and performing tool execution on each split in parallel using native Spark mechanisms, in order to reduce wall-clock time spent performing pipelined data analysis on big data. COMBUSTI/O is not limited to bioinformatics workflows, but may be utilized for any scientific workflow requiring distributed parallel execution of data parallel sequential program binaries in pipelined fashion.

8

Future Work

In this chapter, we discuss possible optimizations and future work for improving COMBUSTI/O, based on the lessons learned and experiences gained implementing and evaluating our framework.

Further Refactoring The implementation of COMBUSTI/O as of the delivery is not of production quality, and would benefit from further refactoring, as the code produced is not as DRY as anticipated given that was one of our design goals. Moreover, the current inheritance-based approach to implementing tool wrappers is likely detrimental to code DRYness as it requires boilerplate code for each wrapper, making the approach using getters and setters enticing. This would entail creating methods extending the current factory method pattern by adding an interface of getters and setters instead of relying on tools being implemented through inheritance from the tool abstraction and existing as a class of its own, which might also increase ease of use.

COMBUSTI/O and Increasing Input Sizes As is, COMBUSTI/O has been evaluate for two extremities in terms of input size, one being tens of megabytes, the other being more than a hundred gigabytes. We recognize the need for an evaluation of how COMBUSTI/O responds to increases in input size in furtherance of establishing a trend, and to see at what sizes issues are introduced, yielding insights in what size applications should be sanctioned and not for use with COMBUSTI/O.

Workflow Language We are looking into supporting the Common Workflow Language (CWL) [90] to help the standardization and portability of workflows definitions, and in turn making it attractive to a broader user base. It is designed for expression of physics, astronomy, bioinformatics and chemistry workflows, and while currently out of the scope in draft 4, it is flexible enough to allow workflow platforms to remotely handle security, cloud, cluster and VM deployment, DFS usage, and identification and recovery of previously executed stages' results.

Dependency Injection Dependencies of bioinformatics tools can be convoluted, and missing dependencies often-times result in pipeline crashes. As a better solution to our validation method, we propose adding support for encapsulation and dissemination of tool binaries and dependencies. One approach to this is using a tool like AppImage [91] for packaging the tools and all their dependencies into a single executable, and distributing the set of all AppImages of tools in a given workflow to all participating nodes prior to running the job, allowing the application to make sure that all dependencies are correct and as a result avoiding unnecessary crashes.

Arbitrary Code Execution Another perspective that is outside the scope of this work is security concerns when forking subprocesses using strings and sequences of strings, being susceptible to code injection not only through our framework, but also being innately susceptible to the vulnerabilities of the tools wrapped. This is to be addressed at several levels, but mainly through extensively sanitizing the input strings before they reach COMBUSTI/O, but tool-specific string restrictions may also be required within the framework.

Execution Isolation Since string sanitation is complex, and the ideal of covering all potential vulnerabilities of implemented tools is improbable, measures for damage control is another aspect that will be addressed in META-pipe from within either COMBUSTI/O or the execution manager. Two approaches are currently under consideration, which are containerization and `chroot`-esque isolation both conducive to mitigating the damage potential of malicious users. Containers provide isolation from the hardware it is run on top of, as well as between individual programs, and there are several ways of achieving containerization—which, unlike virtualization, is done from within the kernel. Docker [92] is one of the contemporaneously popular open-source platforms which may be used with its Swarm extension for cluster management using the same API, and also supports explicit dependency-management; another option is running Spark using the Kubernetes [93] cluster manager to leverage container flexibility, portability, and isolation [94]. The forethought `chroot`-solution isolates the original file system of a node by limiting access through pivoting the current root directory on a per-user or per-workflow basis,

thereby limiting the access privileges of the running process.

Software Enhancements Inspired by the ADAM big data genomics framework, we will be looking to integrate our framework using Apache's Avro and Parquet for extensible binary formats and creating files that may be used directly as SQL tables. Column-based storage formats are preferred over row-based formats for performance benefits of applications that are not write-intensive, and provides high performance when reading [20, 26], making this combination suitable for our primary compute-intensive workloads. Moreover, as pointed out by Nothaft et al. [26], the high compression achieved by the Parquet column-based storage also diminish I/O bandwidth and non-volatile storage consumption, using dictionary encoding that require only three bits per nucleobase for sequence data representation. Encoding formats for nucleotide sequences may involve encoding and decoding to and from formats in between tool stages requiring different input formats, and care should be taken not to trade CPU resources for network and I/O performance improvements too aggressively, as called to attention by Ousterhout et al. [55]. Finally, Nothaft et al. [26] states that their current research efforts partly lie in developing a genome assembler on top of ADAM using GraphX, which is of great interest to our project if it is to support the assembly of metagenomic sequence data, as previous evaluations using META-pipe 1.0 has shown concerning results regarding the scalability of Ray on the Stallo supercomputer [53].

Hardware Enhancements Exploring performance impacts of current and next-generation hardware resources would be interesting, especially experimenting with the Intel and Micron collaboration 3D XPoint non-volatile memory as it is claimed to exhibit exponentially greater durability and significantly lower latency than NAND-based memory technologies, and using SSDs as opposed to mechanic HDDs. Specifically, within stable storage, the Non-Volatile Memory (NVM) Express is an interface for PCI Express SSDs that increases read and write throughput by doing it in parallel, circumvents the latency inefficiencies of using AHCI for SSDs – due to the behavior of SSDs more closely resembling that of DRAM as opposed to slow rotating disks – thus reducing the interaction overhead [95], and it could be worth exploring how it affects the overall performance of, especially data-intensive, tasks in the big data ecosystem.

Source Code

The source code can be found on GitHub (`https://github.com/jarlebass/combustio`).

Bibliography

[1] J. Shendure and H. Ji, "Next-generation DNA sequencing," *Nature biotechnology*, vol. 26, no. 10, pp. 1135–1145, 2008.

[2] S. D. Kahn *et al.*, "On the Future of Genomic Data," *Science(Washington)*, vol. 331, no. 6018, pp. 728–729, 2011.

[3] M. L. Metzker, "Sequencing technologies—the next generation," *Nature reviews genetics*, vol. 11, no. 1, pp. 31–46, 2010.

[4] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big Data: Astronomical or Genomical?," *PLoS Biol*, vol. 13, no. 7, p. e1002195, 2015.

[5] V. Marx, "Biology: The big challenges of big data," *Nature*, vol. 498, no. 7453, pp. 255–260, 2013.

[6] W. Raghupathi and V. Raghupathi, "Big data analytics in healthcare: promise and potential," *Health Information Science and Systems*, vol. 2, no. 1, p. 3, 2014.

[7] Y. Diao, A. Roy, and T. Bloom, "Building Highly-Optimized, Low-Latency Pipelines for Genomic Data Analysis," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR'15)*, 2015.

[8] "EMBL European Bioinformatics Institute." `http://www.ebi.ac.uk/`.

[9] C. E. Cook, M. T. Bergman, R. D. Finn, G. Cochrane, E. Birney, and R. Apweiler, "The European Bioinformatics Institute in 2016: Data growth and integration," *Nucleic acids research*, vol. 44, no. D1, pp. D20–D26, 2016.

[10] "DNA Sequencing Costs." `https://www.genome.gov/sequencingcosts/`. [Online; accessed 12-April-2016].

[11] B. Fjukstad, "Kvik: Interactive exploration of genomic data from the

NOWAC postgenome biobank," 2014.

[12] L. Dai, X. Gao, Y. Guo, J. Xiao, Z. Zhang, *et al.*, "Bioinformatics clouds for big data manipulation," *Biology direct*, vol. 7, no. 1, p. 43, 2012.

[13] "Amazon Web Services (AWS) - Cloud Computing Services." `https://aws.amazon.com/`.

[14] "Microsoft Azure: Cloud Computing Platform & Services." `https://azure.microsoft.com/en-us/`.

[15] "CSC - cPouta." `https://research.csc.fi/cpouta`.

[16] "ELIXIR Data for life." `https://www.elixir-europe.org/`.

[17] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," 2011.

[18] M. C. Schatz, "CloudBurst: Highly Sensitive Read Mapping with MapReduce," *Bioinformatics*, vol. 25, no. 11, pp. 1363–1369, 2009.

[19] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg, "Searching for SNPs with cloud computing," *Genome Biol*, vol. 10, no. 11, p. R134, 2009.

[20] M. Massie, F. Nothaft, C. Hartl, C. Kozanitis, A. Schumacher, A. D. Joseph, and D. A. Patterson, "ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2013-207*, 2013.

[21] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2, USENIX Association, 2012.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, vol. 10, p. 10, 2010.

[23] "Welcome to Apache™ Hadoop®!." `https://hadoop.apache.org/`.

[24] "GitHub apache/spark." `https://github.com/apache/spark/`. [Online; accessed 19-November-2015].

[25] J. Leipzig, "A review of bioinformatic pipeline frameworks," *Briefings in Bioinformatics*, p. bbw020, 2016.

[26] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, *et al.*, "Rethinking Data-Intensive Science Using Scalable Analytics Systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 631–646, ACM, 2015.

[27] "Apache Parquet." `http://parquet.apache.org/`.

[28] "Welcome to Apache Avro!." `https://avro.apache.org/`.

[29] "Node.js." `https://nodejs.org/`.

[30] "Foundation Project." `http://www.apache.org/foundation/`. [Online; accessed 25-November-2015].

[31] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-So-Foreign Language for Data Processing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1099–1110, ACM, 2008.

[32] J. Li, S. Mehrotra, and W. Zhu, "Prajna: Cloud Service and Interactive Big Data Analytics." `https://msrccs.github.io/Prajna/paper/Prajna_v1.pdf`, 2015. [Online; accessed 24-May-2016].

[33] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A Timely Dataflow System," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 439–455, ACM, 2013.

[34] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language.," in *OSDI*, vol. 8, pp. 1–14, 2008.

[35] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10, IEEE, 2010.

[36] "Amazon Simple Storage Service (S3) - Object Storage." `https://aws.amazon.com/s3/`.

[37] "Azure Storage - Secure cloud storage | Microsoft Azure." `https://azure.`

microsoft.com/en-us/services/storage/.

[38] "Distributed File System overview: Remote File Systems; File and Storage Services." https://technet.microsoft.com/en-us/library/cc738688(v=ws.10).aspx. [Online; accessed 16-April-2016].

[39] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem," in *Proceedings of the Summer USENIX conference*, pp. 119–130, 1985.

[40] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.," in *NSDI*, vol. 11, pp. 22–22, 2011.

[41] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 5, ACM, 2013.

[42] "Microsoft HPC Pack." https://technet.microsoft.com/en-us/library/cc514029.aspx. [Online; accessed 16-April-2016].

[43] "Introduction to ELIXIR-EXCELERATE." https://www.elixir-europe.org/system/files/documents/excelerate-introduction.pdf. [Online; accessed 04-May-2016].

[44] "Ray – Parallel genome assemblies for parallel DNA sequencing." http://denovoassembler.sourceforge.net/.

[45] "MetaGeneAnnotator." http://metagene.nig.ac.jp/.

[46] "BLAST: Basic Local Alignment Search Tool." https://blast.ncbi.nlm.nih.gov/Blast.cgi.

[47] "About InterProScan 5 < InterPro < EMBL-EBI." https://www.ebi.ac.uk/interpro/interproscan.html.

[48] "Configuration - Spark 1.3.0 Documentation." https://spark.apache.org/docs/1.3.0/configuration.html.

[49] "Running Spark on YARN - Spark 1.3.0 Documentation." https://spark.apache.org/docs/1.3.0/running-on-yarn.html.

[50] "Job Scheduling - Spark 1.3.0 Documentation." https://spark.apache.

org/docs/1.3.0/job-scheduling.html.

[51] "Spark Architecture | Distributed Systems Architecture." http://0x0fff.com/spark-architecture/. [Online; accessed 13-April-2016].

[52] "Index of pub/databases/uniprot/previous_releases/release-2016_02/uniref/." ftp://ftp.uniprot.org/pub/databases/uniprot/previous_releases/release-2016_02/uniref/. [Online; accessed 11-March-2016].

[53] E. M. Robertsen, T. Kahlke, I. A. Raknes, E. Pedersen, E. K Semb, M. Ernstsen, L. A. Bongo, and N. P. Willassen, "META-pipe - Pipeline Annotation, Analysis and Visualization of Marine Metagenomic Sequence Data." Unpublished (submitted, preprint: http://arxiv.org/abs/1604.04103), 2016.

[54] "Index of /enwiki/." https://dumps.wikimedia.org/enwiki/.

[55] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 293–307, 2015.

[56] J. A. Hoxmeier and C. DiCesare, "System Response Time and User Satisfaction: An Experimental Study of Browser-Based Applications," *AMCIS 2000 Proceedings*, p. 347, 2000.

[57] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[58] "Apache Spark." http://spark.apache.org/.

[59] "AMPLab." https://amplab.cs.berkeley.edu/.

[60] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, ACM, 2010.

[61] "Apache HBase – Apache HBase™Home." https://hbase.apache.org/.

[62] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A Resilient Distributed Graph System on Spark," in *First International Workshop on*

Graph Data Management Experiences and Systems, p. 2, ACM, 2013.

[63] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, *et al.*, "MLlib: Machine Learning in Apache Spark," *arXiv preprint arXiv:1505.06807*, 2015.

[64] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pp. 10–10, USENIX Association, 2012.

[65] J. Goecks, A. Nekrutenko, J. Taylor, *et al.*, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome Biol*, vol. 11, no. 8, p. R86, 2010.

[66] J. Goll, D. B. Rusch, D. M. Tanenbaum, M. Thiagarajan, K. Li, B. A. Methé, and S. Yooseph, "METAREP: JCVI Metagenomics Reports—an open source tool for high-performance comparative metagenomics," *Bioinformatics*, vol. 26, no. 20, pp. 2631–2632, 2010.

[67] "Welcome to the NeLS portal!." `https://nels.bioinfo.no/`.

[68] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, *et al.*, "The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud," *Nucleic acids research*, p. gkt328, 2013.

[69] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, *et al.*, "Initial Sequencing and Analysis of the Human Genome," *Nature*, vol. 409, no. 6822, pp. 860–921, 2001.

[70] "Feide." `https://www.feide.no/`.

[71] "Big Data Genomics." `http://bdgenomics.org/`.

[72] A. J. Vander and D. JH Luciano, "Vander's Human Physiology: The Mechanisms of Body Function," 1980.

[73] E. Pettersson, J. Lundeberg, and A. Ahmadian, "Generations of sequencing technologies," *Genomics*, vol. 93, no. 2, pp. 105–111, 2009.

[74] J. Handelsman, J. Tiedje, L. Alvarez-Cohen, M. Ashburner, I. Cann, E. De-

long, W. F. Doolittle, C. Fraser-Liggett, A. Godzik, J. Gordon, *et al.*, "The New Science of Metagenomics: Revealing the Secrets of Our Microbial Planet," *Nat Res Council Report*, vol. 13, 2007.

[75] "Web BLAST page options." http://blast.ncbi.nlm.nih.gov/blastcgihelp.shtml. [Online; accessed 8-March-2016].

[76] P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice, "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants," *Nucleic acids research*, vol. 38, no. 6, pp. 1767–1771, 2010.

[77] Q. Zhang, J. Pell, R. Canino-Koning, A. C. Howe, and C. T. Brown, "These Are Not the K-mers You Are Looking For: Efficient Online K-mer Counting using a Probabilistic Data Structure," 2014.

[78] P. E. Compeau, P. A. Pevzner, and G. Tesler, "How to apply de Bruijn graphs to genome assembly," *Nature biotechnology*, vol. 29, no. 11, pp. 987–991, 2011.

[79] S. Boisvert, F. Laviolette, and J. Corbeil, "Ray: Simultaneous Assembly of Reads from a Mix of High-Throughput Sequencing Technologies," *Journal of Computational Biology*, vol. 17, no. 11, pp. 1519–1533, 2010.

[80] H. Noguchi, T. Taniguchi, and T. Itoh, "MetaGeneAnnotator: detecting species-specific patterns of ribosomal binding site for precise gene prediction in anonymous prokaryotic and phage genomes," *DNA research*, vol. 15, no. 6, pp. 387–396, 2008.

[81] H. Noguchi, J. Park, and T. Takagi, "MetaGene: prokaryotic gene finding from environmental genome shotgun sequences," *Nucleic acids research*, vol. 34, no. 19, pp. 5623–5630, 2006.

[82] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[83] P. Jones, D. Binns, H.-Y. Chang, M. Fraser, W. Li, C. McAnulla, H. McWilliam, J. Maslen, A. Mitchell, G. Nuka, *et al.*, "InterProScan 5: Genome-scale Protein Function Classification," *Bioinformatics*, vol. 30, no. 9, pp. 1236–1240, 2014.

[84] "Tara Oceans science - EMBL." http://www.embl.de/tara-oceans/start/. [Online; accessed 17-February-2016].

[85] S. Pesant, F. Not, M. Picheral, S. Kandels-Lewis, N. Le Bescot, G. Gorsky, D. Iudicone, E. Karsenti, S. Speich, R. Troublé, *et al.*, "Open science resources for the discovery and analysis of Tara Oceans data," *Scientific data*, vol. 2, 2015.

[86] "ELIXIR Scientific Programme 2014-2018." http://www.elixir-europe.org/system/files/elixir_scientific_programme_1.pdf. [Online; accessed 22-November-2015].

[87] "ELIXIR and ELIXIR-Norway — Site." http://www.bioinfo.no/elixir. [Online; accessed 11-February-2016].

[88] "Molecular Biosystems Research Group (MBRG) | UiT The Arctic University of Norway." https://en.uit.no/forskning/forskningsgrupper/gruppe?p_document_id=349423.

[89] "Biological Data Processing Systems (BDPS)." http://bdps.cs.uit.no/.

[90] "Common Workflow Language." http://www.commonwl.org/.

[91] "AppImage | Linux apps that run anywhere." http://appimage.org/.

[92] "Docker - Build, Ship, and Run Any App, Anyhwere." https://www.docker.com/.

[93] "Kubernetes - Accelerate Your Delivery." http://kubernetes.io/.

[94] "Kubernetes: Using Spark and Zeppelin to process big data on Kubernetes 1.2." http://blog.kubernetes.io/2016/03/using-Spark-and-Zeppelin-to-process-Big-Data-on-Kubernetes.html. [Online; accessed 19-May-2016].

[95] D. Landsman, "AHCI and NVMe as Interfaces for SATA Express™ Devices - Overview." SanDisk https://www.sata-io.org/sites/default/files/documents/NVMe%20and%20AHCI%20as%20SATA%20Express%20Interface%20Options%20-%20Whitepaper_.pdf. [Online; accessed 24-May-2016], 2016.