Workflow Orchestration On WeevilScout

Ganeshwara Herawan Hananda Putra

March 4, 2013

Contents

Abstract **2**

Foreword **3**

1 Introduction **4**

1.1 Motivation 4

2 Background **6**

2.1 Concurrent Systems 6

2.1.1 Models Of Concurrency 6

2.1.2 Concurrency In Computer Systems 9

2.2 Distributed Systems 10

2.2.1 Types of Distributed Systems 10

2.3 Scientific Workflow Management Systems 12

2.3.1 The Grid Workflow System 12

2.3.2 Models of Computation 13

2.3.3 Service Orchestration & Service Choreography 16

2.4 Web Technologies 17

2.4.1 Browsers 17

2.4.2 ECMA Script / Javascript 17

2.4.3 WebWorker 18

2.4.4 WebCL 19

2.4.5 Browser Security 19

2.5 Related Works 20

2.5.1 Ravan 20

2.5.2 MapRejuice 22

2.5.3 BOINC 23

2.5.4 Traditional SWMS Platforms 24

3 WeevilScout Architecture **26**

3.1 High Level Architecture 26

3.2 Work Unit 28

3.2.1 WebWorker Job Description . . . 29
3.2.2 WebCL Job . . . 30
3.3 Workflow . . . 31
3.3.1 Workflow Structure . . . 32
3.3.2 Workflow Model . . . 33
3.4 Workflow Description Language . . . 37
3.4.1 Elements . . . 38
3.4.2 Parsing Steps . . . 42
3.5 Scheduling and Execution . . . 46
3.5.1 Alternatives . . . 47
3.6 Dataflow Scheduling Using Actor Model . . . 50
3.6.1 Root Supervisor . . . 50
3.6.2 Executors . . . 50
3.6.3 Messages . . . 51
3.6.4 Executor Types . . . 52
3.7 Job Queueing . . . 56
3.7.1 Implementation . . . 57
3.8 Execution In Remote Workers . . . 58

4 Evaluations **60**
4.1 Metrics . . . 60
4.2 Profiling Framework . . . 60
4.3 Platforms . . . 61
4.3.1 SARA HPC Cloud . . . 61
4.3.2 Intel Core2 Duo PC . . . 61
4.4 Initial Benchmarks . . . 62
4.5 WeevilScout Performance Evaluation . . . 64
4.5.1 Attained Performance Test . . . 64
4.5.2 Pi-Approximation Test . . . 65
4.5.3 Probing For The Optimal Sample Size . . . 66
4.5.4 Results From Intel Core2 Duo PC . . . 68
4.5.5 Results From SARA HPC Cloud . . . 70

5 Conclusions And Future Works **74**
5.1 Conclusions . . . 74
5.2 Future Works . . . 75

6 Appendix **77**
6.1 Job Examples 77
6.2 Workflow Examples 79

Abstract

The proliferation of web-browsers coupled with emergence of HTML5 and recent advances in Javascript engine from major browser vendors have opened the door to distributed computing on web-browsers. This paper describes an enhancement to WeevilScout, a new volunteer computing-based, scientific workflow management system that allows for execution of jobs on an ensemble of browsers in parallel. WeevilScout utilizes two core technologies that have yet to see mainstream adoption, the WebWorker which allows for Javascript execution in the background and WebCL which provides direct hardware access from within the browser. A workflow coordination engine that we have developed allows for defining inter-job dependencies by function-level paralellism. The result showed that we were able to gain considerable amount of computing power when there were enough users connected to the system.

Foreword

I would like to thank the following people, without whose help and support this thesis would not have been possible. Prof. Adam S.Z. Belloum as my thesis supervisor, for his guidance, encouragement, and support of the thesis project and the years of study in general. Reginald Cushing as my thesis supervisor, for his vision, suggestion, and continued support of my thesis project. My lecturers and tutors that had provided continuous support during the years of study at the Universiteit van Amsterdam.

Finally I would like to thank my parents and friends for their constant support and encouragement during the course of the thesis project.

Chapter 1

Introduction

The abundance of data produced during a scientific experiment underlines the need for a fast and effective means of data processing. In practice this often means performing experiments on a distributed system, where scientific data that are often concurrent by nature can be processed by an array of computing resources.

In grid computing, scientific experiments are carried by multiple collaborating organizations on an array of possibly heterogeneous computing resources that span across multiple administration domains.

In another scenario where there is not enough available computing resources, one can resort to volunteer computing[44], where anyone with some CPU cycles to spare can participate in the experiment by letting their CPU be used to perform the experiment.

1.1 Motivation

While distributed computing allows for harnessing a large amount of computing power, the complexities that are involved in building and maintaining it are notorious. As a response, various middlewares have been developed with the aim of abstracting away the technical details. While these middlewares have generally been successful to some extent, they are often riddled with various configuration settings that again requires a degree of technical know-how.

This thesis report describes an enhancement to the new volunteer computing-based, scientific workflow management system that allows for execution of jobs on an ensemble of browsers in parallel. Dubbed the WeevilScout, our main motivation is to provide a platform that is very quick to set-up and deploy while lowering the entry point barrier, achieved via the use of high-level language as well as intuitive dataflow-oriented workflow model.

A feature unique in WeevilScout that differentiates it from most conventional scientific workflow management systems is that jobs are written in Javascript and are meant to be executed on the web-browser running on volunteers' machine.

With 2 billion users online, browser computing has the potential of amassing immense resources necessary for scientific experiments.

Also, in web-browsers we have come to recognize a powerful yet unexplored feature, that is, their ability to provide a uniform execution environment that hides the heterogeneity of the underlying platforms.

Volunteers can participate in a project by following a single step that consist of accessing the project URL in their browser. We believe such ease of access will be a major virtue that will accelerate the adoption rate of our platform by online users.

Finally, there have indeed been many platforms that aim to extract computing powers from complex, distributed environment for scientific purposes. However as elaborated above, we believe that WeevilScout has the potential to fill in the gap left open by the existing platforms.

Chapter 2

Background

This chapter elaborates a few closely related concepts relevant to our work in WeevilScout. Most importantly, it elaborates important concepts such as the actor model, dataflow execution model, WebWorkers, and WebCL. The last section in this chapter elaborates some of the most prominent works in the field of web-based distributed systems and scientific workflow management systems.

2.1 Concurrent Systems

A system is said to be concurrent if it allows for the execution of more than one *activity* simultaneously. Concurrency can happen at many levels in a computer system. For example, in a multi-processor system concurrency is achieved by the means of *threads* and *processes*. In a *distributed system* such as BOINC[4] or WeevilScout, concurrency refers to how activities are performed in more than a single machine simultaneously.

2.1.1 Models Of Concurrency

In describing a certain phenomenon, formalizing important behaviors are often beneficial in order to gain a better understanding of it. As such there have been ongoing efforts in formalizing concurrent systems using a variety of models.

Process Algebra and *PetriNet* are two approaches in modeling concurrency which had been considered for the dataflow-based scheduler in WeevilScout. However, we decided that it should be implemented with the *actor model*. The motivation for such a choice is described in the next subsection.

Process Algebra refers to a family of approaches of modeling concurrent systems using a rigorous mathematical framework[14], and some of the most prominent works includes Calculus of Communicating Systems[35], Algebra of Communicating Processes[9], and Communicating Sequential Processes[28]. A process is an entity which runs sequentially and concurrency is achieved by the execution of

several processes. Process algebra focuses on processes and their communication, which are commonly represented with well-known mathematical symbols.

Petri Net[39] is a graphical notation that models state and transition of some systems. Mathematically, a Petri Net N can be described as a 3-tuple, *(P, T, F)* where P denotes the set of possible states, T the set of transitions and F the arcs that defines the relation between a place p and a transition t. A Petri Net is also a *bi-partite* graph with set *(P, T)* connected by F[1], meaning that an arc can only connect two nodes of differing types, such as a node p from set P to a node t from set T, and vice versa.

Petri Net has seen wide range of usage in various fields such as business process management[51], manufacturing[20] and computer science[53] and the original concept has been modified and extended[34] many times to fit into domain-specific use-cases. Not limited to the aforementioned, Petri Net has also been used for workflow coordination in grid computing, such as in *GWorkflowDL*[3].

Actor Model

Process Algebra and PetriNet both provide sensible ways of modeling concurrency. However we feel that the actor model is best suited for modeling our dataflow-based scheduler. The main reason is that actor model puts emphasis in modeling a set of independent, concurrent objects, each with its own set of behaviors. For example, we did not see the merit of modeling our problem as a set of states and transitions in PetriNet. Moreover, the environment we used to program WeevilScout already provides a mature implementation of the actor model.

Actor model represents a system in terms of *actors*. Actors are *autonomous*; they should be able to operate based on *local information*, make *local decisions* and change their *local states* accordingly[2]. Communication between actors are done via *messages*[27].

There have been an effort to provide synchronization in actor models[6], but in the context of this project, the *ordering* of arriving messages is non-deterministic and thus should *never* be allowed to affect how an actor changes its local state[21] [15].

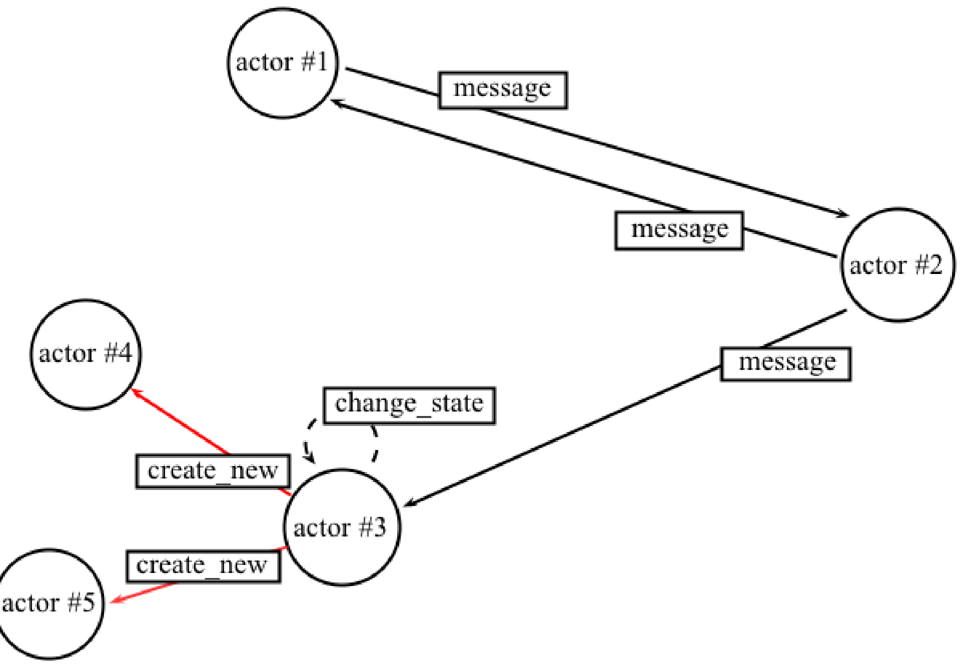

Figure 2.1: Activities and communications between actors

Figure 2.1 shows the typical activities between actors such as sending messages and changing internal states. Also, an actor can spawn new actors and remove them[15]. Thus during a system's lifetime, new actors can be created and those who no longer serves any purpose destroyed.

Actor model can be used to model various kind of systems. In a system that runs on a multi-core processor, an actor might represent an entity that performs and manages a concurrent computation where it might send the result to some other actors afterwards. While waiting for this result, these actors might also perform other computations in parallel to the activities performed by the first actor.

In another scenario involving a distributed system that runs on machines spread across remote areas, an actor might represent a single machine and does things such as managing how it should exchange information across the network with the others.

A few programming languages which are designed for creating concurrent applications such as Erlang and Scala supports the actor model through it's standard library.

During the thesis project, we found that actor model allowed us to describe our dataflow-based workflow naturally. In our approach, a workflow structure is represented as a DAG graph and during workflow execution, each vertex will

be represented by an actor, and edges between vertices are the only allowed communication-channels between actors[1].

2.1.2 Concurrency In Computer Systems

The previous section described the different approaches to reason about concurrency at a certain abstract level. In practice however, concurrent execution in computer systems usually happen at one of the three levels: *instruction level*, *thread level* or *process level*.

At the time of writing, current machines and compilers are generally very bad at inferring concurrency in a program, and so programmers must use means such as *threads* or *message passing* to explicitly define the concurrent behavior of the said program.

There have been effort to raise the abstraction level in some languages. For example, actor model concurrency is supported in Smalltalk, Erlang and Scala language, amongst others. In Scala's actor model, concurrency is achieved by assigning an actor instance a thread from a *thread pool* that is implicitly managed by the Scala runtime. Another example is the Single Assignment C (SAC)[45], a functional-programming language with a focus on array processing which allows for implicit parallelism to a certain extent.

Threads & Processes

In WeevilScout, multiple threads will be spawned when actor instances are created upon the execution of a workflow. Threads are created and managed using a *thread pool*, and each actor will be assigned to a thread at the time of execution. The thread pool is generally managed by the Scala runtime, but various parameters can be inserted to guide its behavior. This section is meant to refresh the readers of the basic concept of threads and processes.

A *thread* is an independent execution point within a process that can be executed concurrently. Generally multithreading has been used to allowing for concurrent execution which will benefit performance in two ways. First, execution will be done in parallel if the host contains a multi-core processor. If this is the case, performance will increase by the factor of n, which is the number of available cores assigned to the said threads. Second, even if the system only has a single-core processor, performance can still benefit from multi-threading, as it can mitigate the effect of I/O blocking that often happen when a program reads from

[1]See Section 3.3 and 3.6

the memory or disk. This is possible since in a single-core processor, threads are still scheduled using means such as time-division multiplexing. In that case, a thread can be blocked temporarily when it access memory or disk, thus allowing other threads to execute in the meantime.

Processes, like threads, allows for concurrent execution. This behavior is commonly exhibited in most modern operating system, which allows multiple processes (or program instances) to be opened at the same time. Process communicates using various Inter-process Communication methods defined in the host operating systems. They allow process to read from and write to a shared space (shared memory), exchange message (message passing) or call subroutine that exist in foreign address space (RPC).

2.2 Distributed Systems

As an application where tasks are offloaded to many machines, WeevilScout can be classified as a distributed system. However, in contrast to typical distributed systems where a software component must be present on all machines that are part of the system, execution of tasks are done within the web-browser running on those machines.

A distributed system refers to an architecture where programs are running on multiple machines, connected via low-speed communication channels such as the Local Area Networks. In distributed systems, the machines are often heterogeneous, faulty, and typically do not have complete information of the global system state.

In practice, practical constraints such as machine heterogeneity, reliability and even locations often dictates the structure of a particular distributed systems. For example, a system running through machines that are connected via high latency networks will have to be structured to mitigate its negative effect. A system that runs on machines with a high failure rate have to be designed to tolerate these failures.

2.2.1 Types of Distributed Systems

There are several scenarios of which a distributed systems can be arranged. For example, *volunteer computing* is a distributed computing scenario where computing resources are donated voluntarily by willing participants with available

computing resources. WeevilScout, for example, is an example of this type of distributed system, since execution of tasks are offloaded to the browsers running on the machine of volunteers.

Volunteer computing are often compared with *grid computing*. In grid computing, the collaborating organization will be benefitted directly when the goal is fulfilled while in volunteer computing, volunteers donate computing power mostly for altruistic reason (although a reward mechanism does exist in some middlewares[4]).

Like in grid computing, volunteer computing utilizes a middleware to manage the available resources, which in this case are machines owned by the volunteers. The middleware usually consist of two main part. The central part which coordinates activities that usually resides in the host of the project owner and the client part which must reside in all of the volunteers' machine. In addition, a volunteer computing project usually provides a spot for the community, such as a forum or mailing list for keeping contact and maintaining the enthusiasm of volunteers.

In *cluster computing*, a cluster is made up by a set of computing nodes which communicate via a dedicated network interconnect. The nodes in a cluster are often made up from a set of fully operational computing machines, each consisting of an independent CPU, memory, and storage devices[7, Ch. 1 p. 6]. Cluster computing paradigm has been used by many, from financial, manufacturing and scientific institutions as a means of powering high-performance computation at the fraction of cost of building a dedicated supercomputer[7, Ch. 1 p. 3]. The advent of various programming tools such as the message passing interface (MPI) as well as various message-queueing library (RabbitMQ, JMS, ActiveMQ, and so on) have long aided the creation of software systems that run on a cluster setting.

Service-Oriented Architecture (SOA) is another approach towards building a highly scalable distributed system. As have been made apparent in grid computing, the heterogeneity of computing resources have complicated the process of building a distributed system. The driving motivation behind SOA is to hide the details of implementation by encapsulating them into a *service*. A service may represent a process, a database access, or an output from a sensor, amongst other things. The goal of SOA is to relieve the burden of dealing with low-level intricacies from the service *consumer*, which is someone who makes use of a particular service. A service should be loosely-coupled to allow for system reconfiguration by exchanging bits and parts within the system.

2.3 Scientific Workflow Management Systems

Task coordination in WeevilScout is realized using workflow approach. However, the concept itself is not new. The workflow approach that is facilitated by the use of workflow management systems such as BPMN[57] have had its practical use in the business field, where it is commonly used to automate various business processes[58].

Recently, the workflow approach have been adopted by scientists for coordinating scientific experiments. However, it was plagued with plethora of proprietary systems that failed to provide benefits for the scientific community. Fortunately there have been a significant amount of work in bringing open platforms for the scientific community[33][59]. These platforms, dubbed the *Scientific Workflow Management System* (SWMS) are meant to be open, interoperable and highly configurable while allowing coordination of scientific activities.

As a distributed application where coordination of scientific tasks are done using workflow, WeevilScout could be classified as another instance of SWMS.

2.3.1 The Grid Workflow System

In [59] the author argues that most scientific workflow system that runs on the grid comprises of components as will elaborated in the next few sub-sections.

Workflow Design

Workflow design refers to the rules and constraints that concerns about how components of a workflow might be defined and composed[59].

Workflow Structure

A workflow structure defines the inter-relation of tasks. It might allow for structural constructs such as choice, sequential or parallel execution. In general, a workflow structure might be based on directed acyclic graph. a DAG-based workflow does not allow iterations or loops which are, in contrast, supported in a non DAG-based workflow.

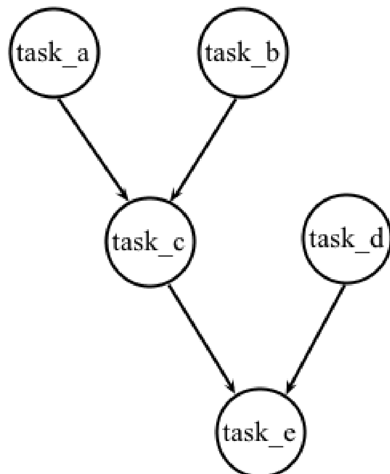

Figure 2.2: A DAG-based workflow structure

Figure 2.2 depicts a workflow consisting of 5 tasks. The arrow denotes dependencies between task, imposing sequential execution constraints. Parallel execution is *inferred*, not explicit. In that instance, task_a, task_b, and task_d might be executed in parallel as they do not depend on other task(s). task_c and task_d may also be executed in parallel, provided that task_a and task_b have sent their results to task_c.

Workflow Model

A workflow model is made of both structural definition as well as the tasks that it contains, or more formally, a set of tasks T and a set of edges E which defines the dependencies between each task $\{t_1, t_2, t_3, ..., t_n\} \in T$. Again [59] argues that there are two types of workflow model. In the *abstract model*, details of how tasks should be executed is abstracted away while in the *concrete model*, these details are fully-exposed.

Abstract model have the advantage of delaying the assignment of tasks to resources as late as possible, and potentially make use of run-time information for optimal assignment of tasks. This is not the case for concrete model, since resource assignment is decided before execution. Often, a combination of abstract and concrete model can be used, meaning that partial information about resource mapping is written as part of the description, where the rest can be determined during an execution.

2.3.2 Models of Computation

If a workflow model is a definition that consists of a set of tasks T and a set of edges E which defines the dependencies between each task $\{t_1, t_2, t_3, ..., t_n\} \in$

T, then a model of computation (MoC) M of that particular workflow defines a formal abstraction of how it should be executed by a particular SWMS in a computer system.

Different MoCs offer different ways of expressing a scientific workflow, and some types of MoCs might fit a particular scientific experiment better than the other. In addition, composing a workflow which are executed under a combination of several MoCs can also prove to be useful, e.g integrating devices such as sensor networks or electron microscopes with continuous dynamics, into a dataflow-based workflow model[22]. Kepler, for instance, supports multiple MoCs.

The following sub-sections elaborates some of the most commonly used MoCs.

Process Networks

In process networks (PN), tasks are described as actors which can be executed concurrently. An actor have input and output ports, which are places for it to receive data from, and send results to. The number of available ports are defined in the workflow definition. During its lifetime, an actor will read tokens containing input data arriving at one of its input ports. It will perform computation when it receives enough input and writes the result to its output ports.

Dataflow

In dataflow, an actor is brought to life when all of its input are available. More specifically, in dynamic dataflow (DDF) various run time parameters are considered when scheduling an actor while in static dataflow (SDF), only static information which are available from the start are considered.

The execution model of WeevilScout itself falls under static dataflow; it schedules an actor when all of its required input are available. Moreover, it does not (yet) consider any run time information for selecting which actors to be scheduled for execution.

Discrete Events

In discrete events (DE), actors communicate using tokens that contains data. Each token is associated with a timestamp of when communication occurred i.e. when an output is sent by a producer to its consumer. An actor is fired when one or more of its input ports contain the oldest token out of all unconsumed tokens, or when it specifically requests to fire at a timestamp less than the timestamp present in all of the unconsumed tokens and the timestamp of request to fire from other actors[23].

Synchronous/Reactive

In synchronous/reactive (SR), a global clock is maintained and every actor is fired at each tick. When an actor is fired, it observes input values and asserts output values. An important property to maintain is that even when an actor is repeatedly fired, if the all inputs remain identical to how they were on the previous firing, then the asserted output must also remain the same.

Workflow Scheduling

Before a scientific task can be scheduled and executed, an SWMS will need to decide on task priority. Some factors must be considered, such as task dependency and the required resource for executing a particular task. Scheduling can either be *centralized*, *decentralized* or *hierarchical* and the choice of which strategy to implement need to be carefully assessed as it will impact scalability and performance[26].

Centralized

A centralized scheduler makes the decision for all tasks. As such, it will always be able to calculate the most optimal scheduling since having access to all tasks enables it to collect all the necessary information. However it is not scalable, as a large number of tasks at hand might overwhelm the scheduler.

Decentralized

In a decentralized strategy, a number of local scheduler makes decision only for the tasks of which they are assigned to. As they have limited access to information, they may never calculate the most optimal scheduling for a particular case. However this mechanism is scalable, as an increase in the number of tasks can be dealt by increasing the number of schedulers.

Hierarchical

Hierarchical scheduling somewhat provides a middle ground for centralized and decentralized scheduling. With this approach, a hierarchy of schedulers are maintained and different types of schedulers can be employed at different positions in the hierarchy. The downside of this structure is that it takes some effort in maintaining the hierarchy structure. Also, a failure in one scheduler instance might affect others which depend on it.

Fault Tolerance

As distributed systems are faulty by nature, a measure of dealing with fault is always critical. Typical failure in a distributed system are crashes and loss of connections. During a failure, part of the system must be able to decide if connections are lost, or if a particular node crashes or simply busy for a short amount of time. Then, an appropriate response can be formulated and executed, such as whether to kill, restart, or even to leave it as it is (which can be desirable at times).

Various strategies can be employed to mitigate failure. For example, state checkpointing can be performed periodically on critical nodes so that if it crashes, computation does not have to be restarted from zero.

Part of the system might be faulty and produces incorrect results. This kind of errors are subtle and therefore difficult to detect. Results will need to be checked for correctness, and if there is an indication that a particular result is erroneous, the task which produces it can be in multiple nodes. Then, the results are gathered and the correct result decided by various means, such as voting.

Data Movement

In a grid infrastructure, the execution of scientific tasks are often scheduled to happen at various resources in different locations, and so there will be a need for a strategy of moving data the producer to consumer efficiently. Again, in [59], it is argued that there are three main approaches for data movement strategies.

In the *centralized* approach, a single point of transfer acts as the mediator, or in other words the transit point for every data that needs to be moved. In the *mediated* approach, data transfers are managed by some distributed data management system that decides the most optimal way for moving data from the source to destination. In the *peer to peer* approach, data is transferred directly between the requesting and serving tasks.

As usual, the principal of distributed systems apply here: centralized approach will be the simplest to develop and maintain but is not very scalable, while mediated and peer-to-peer approach offers better scalability but makes the system more complex to develop and maintain.

2.3.3 Service Orchestration & Service Choreography

Service orchestration[38] refers to workflow systems that are coordinated centrally by some kind of a head controller. The lower service which does the actual

work does not have the whole picture of which it is being a part of. BPEL[19] and YAWL[52] are some implementations of this approach. Acting as the head controller, the description controls the flow of data and when to execute processes.

Meanwhile, service choreography[38][13] refers to a system that utilizes decentralized coordination approach. Every component is considered equal, which is in contrast to service orchestration where there is a clear distinction between a coordinator and those who work for it. The WS-Choreography specification developed by the W3C is an example of service choreography for web-services.

2.4 Web Technologies

Over the past decade, the web have moved at such a blistering pace. What used to be a simple medium for serving static contents is now capable of various rich media contents as well as interactive real-time applications. This section describes some of the technologies that are key to WeevilScout.

2.4.1 Browsers

Browser have arguably been at the forefront, a major contributor to the web's growth over the past years and in the post browser-wars era[12] of today, have become much better at adhering to standards. As one of the most ubiquitous applications, people have tried to break the barrier of what it can do. What started as a simple application to display static contents have now evolved significantly, even supporting its own scripting environment as well as being able to support various kind of rich media contents. It has been shown that there is a growing motivation of enabling even more serious class of applications in browsers shown by works such as in SpiderMonkey[37], V8[24], WebGL[32], and WebCL[29].

2.4.2 ECMA Script / Javascript

Javascript allows for dynamic interaction within a web-page and has seen much wider adoption than it was a decade ago. Long ago, its use was limited to adding a layer of interactiveness over a mostly static web-page, mostly due to the speed limit in both interpreter that executes it and most consumer-grade CPU meant to run it. However CPU have gotten much faster and have allowed a higher degree of parallelism with the advent of multi-core chips. Browser vendors, too, have put emphasis on the speed of execution to allow much richer experiences than in the past. The advent of dedicated Javascript engines such as Google

Chrome's V8, Mozilla Firefox's SpiderMonkey and Internet Explorer's Chakra provided significant speed boost in Javascript by implementing various techniques such as just-in-time compilation, code transformation and optimization, and inline caching, amongst others.

The latest addition to Javascript is the WebWorker (explained in the following sub-section) which allows for true parallelism in web-browsers.

2.4.3 WebWorker

WebWorker allows for parallel execution of Javascript. It is implemented as a thread, but does not allow the shared memory model and instead relies on message-passing which in Javascript is implemented using its event listeners model.

Figure 2.3: Communication model in dedicated workers (left) and shared workers (right)

Figure 2.3 shows the difference in communication model between *dedicated* and *shared* workers. Dedicated workers only allow for communications between the main page and the workers it spawns, while shared workers allow communication between an arbitrary worker to another. WeevilScout, for instance, utilizes the former for executing Javascript jobs.

Moreover, communications can only happen if they all come from the same origin[18]. Furthermore, access to the DOM elements of the web page are restricted, although read-only accesses to some of the `window.*` properties are allowed.

At the time of writing, the WebWorker are still in CR[54] status, meaning that it is not yet part of the standard[55], however it does show promising signs of activity.

2.4.4 WebCL

WebCL, an emerging standard developed within the Khronos Group, is a Javascript binding to OpenCL[47], providing web applications direct access to heterogeneous parallel computing resources such as the CPU or GPU available in the client machine[29]. Direct hardware access means that execution can achieve significantly higher throughput than a normal Javascript program does. However this does come at the cost of significantly higher code complexity. A WebCL code is essentially OpenCL code embedded within Javascript, thus burdening the programmer of having to program in two completely separate environment. The only interface between a WebCL code and Javascript is a range of functions for inputting parameters and queries the kernel for result. Furthermore, OpenCL, unlike Javascript, is a static-typed language that more resembles C in its *rigidness*, which might not be received very well by the Javascript community that are used to the flexibility of Javascript syntax.

Last, WebCL is still in its infancy and currently only supported via experimental, 3rd party plugins that is available only on a handful of browsers[49][43].

2.4.5 Browser Security

As one of the most used application on desktop and mobile computers, the browsers have become one of the most targeted applications by attackers. It is an inherently complex application which requires significant efforts in order to keep it secure.

3rd party plugins such as Adobe Flash and Java have contributed to a large amount of security problems. Often, a successful attack comes from tricking the user into thinking that he/she is visiting a safe place, such as by disguising an attack site to look identical to a safe and credible ones. Another common scenario is attacking an outdated browser software with a known security hole.

The lesson learnt is that there are no bullet-proof ways of making the browser completely secure from any kind of attacks. However, there are various techniques that can and should be employed by vendors in order to make their browser more secure. For example, the *same-origin policy* prevents web-sites from different domains to interact with each other by restricting scripts from running on pages that do not come from the same *origin*[42][8]. Moreover, the process of updating the software must be made as pain-free as possible, as security hole are often introduced in user who is reluctant to perform updates frequently.

Last, a sandbox environment is often introduced as a shield that limits potential damages from vulnerabilities in a component that interacts with outside world. A particular example of such component is the rendering engine, which converts a textual representation (that might be infiltrated by malicious code) of a web-page into its graphical representation. In Google Chrome, the rendering engine is placed within a low-privilege sandboxes consisting of three layers, Javascript sandbox, OS/runtime exploit barriers and OS-level sandbox, severely limiting communications to external modules[41].

2.5 Related Works

This sections describe some of the most popular scientific workflow management systems. However, aside from a few discussions[25], there have been little actual activity in the field of web-based distributed systems. However the works on Ravan and MapRejuice are worth mentioning.

2.5.1 Ravan

Ravan[30] is a web-based, distributed hash-cracking application. The Ravan server will take a hash code along with salt, hashing algorithm and character-encoding used as inputs. After they are received, it will return an URL of a web-page that contains a piece of Javascript code that, when executed by clients, will perform the actual brute-force algorithm on the hash that was part of the input. The algorithm is executed in the background thread using WebWorker. Aside from generating code and page links, the Ravan server also checks for the validity of input received from the client where actual brute-forcing operation is performed at.

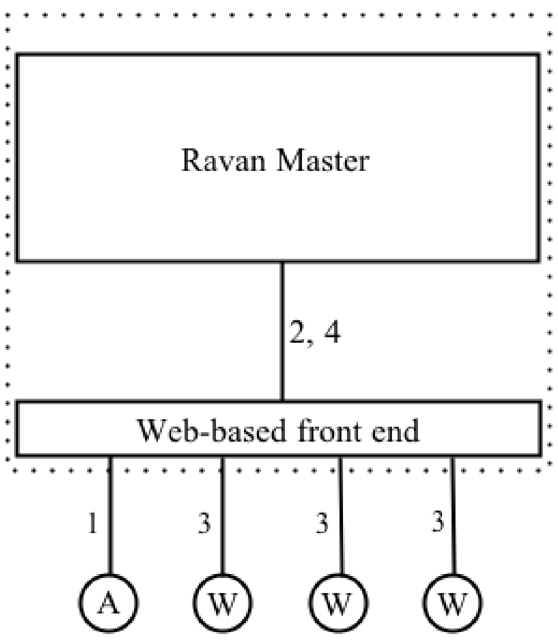

Figure 2.4: High-level Architecture of the Ravan tool

Figure 2.4 describes the lifecycle of a typical distributed hash-cracking activities using Ravan. The *worker farm*, which refers to computing resource on which the hash-cracking operation will be performed on consist of a set of client machines a.k.a the *workers*, that loads the Ravan web-page. In 1) an administrator submits a new hash along with various parameters such as salt and the algorithm used to the web-based front end. 2) Upon receiving these data, the *Ravan master* will create an array of *slots* and pass them to the web-based front end. A slot represents a part of the key space of 1 million combinations that can be ran concurrently by the workers. In 3) worker(s) access the URL of a specific hash that was submitted and will be allocated an existing slot. After this slot has been processed, the worker submits the result back and the slot will be marked as complete. 4) Ravan master continually checks if all slots have been processed and will keep on allocating a new set until the hash is cracked.

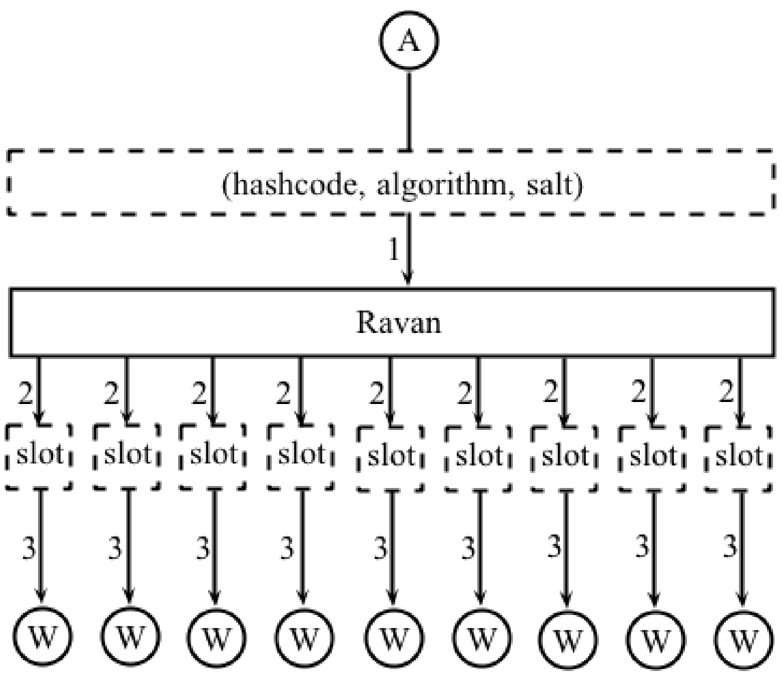

Figure 2.5: Data movement in Ravan

Figure 2.5 describes in more detail how hash is being distributed to the workers. The box with dashed line depicted the actual data where the box with solid lines represents the Ravan system which is depicted as a *black box*. The circles represented external entities, the administrator and the workers. In 1) an administrator submitted a hash code with parameters such as the algorithm and salt. In 2) The data was splitted into slots where 3) each one would be processed in a worker.

2.5.2 MapRejuice

Another work in this field is MapRejuice[31]. MapRejuice is a web-based application which allows execution of an arbitrary algorithm on data input with an execution model that takes inspiration from Google's MapReduce[60]. This is in contrast with Ravan which is specific to hash cracking with algorithm already embedded in it. However it also utilizes WebWorker to execute Javascript code in the client-side just as Ravan does.

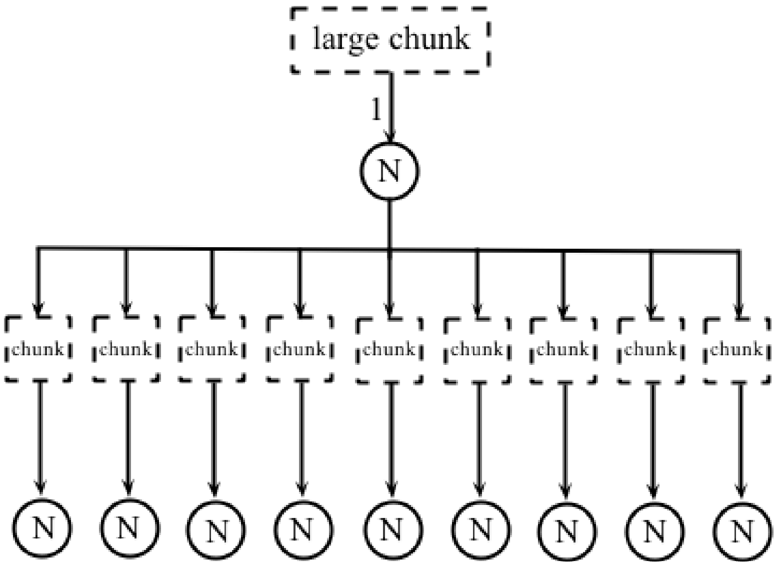

Figure 2.6: Data movement in MapReduce

The original MapReduce design puts emphasis on the ability to process large number of data items on a collection of computing resources, or *nodes* with the goal of exploiting parallelism in the algorithm. Upon execution, the master node maps the input and 2) divides them into an array of smaller chunks and send them to other nodes, or the *children*. The children, in turn, could divide them further into an even smaller chunks and deliver them to more nodes, in essence creating a tree-like structure of nodes. During the *reduction* operation, a node collects and reduces the result from its children and deliver the result to its parent until it reach the top of the tree.

2.5.3 BOINC

BOINC is one of the most popular volunteer computing platform that was developed at the University of California in 2002. It is used in various scientific projects such as SETI@home, Einstein@home, Climate@home, Climateprediction.net, and Predictor@home amongst others, with aggregated performance close to 8 petaFLOPS[10] from volunteers around the world.

A project owner can create a BOINC project which can be made of one or more applications, and during a project lifetime new applications can be added or existing ones removed. The BOINC server stores project details such as application description, version, work units, results, and so on. Scheduling servers issues works and receives report of completed results and data servers handles file uploads[5].

Volunteers can join a BOINC project by visiting the web-site to register. In contrast with our WeevilScout which does computation within a web-browser, a client application that queries and executes work will need to be installed on the volunteer's machine. The client software can be used for joining multiple project i.e. there is no need to install a client for separate projects.

2.5.4 Traditional SWMS Platforms

Established SWMS platforms have greatly influenced the design of WeevilScout by demonstrating that the workflow approach is suited for coordinating of a large number of activities to be performed on a complex environment such as the grid. This section contains some of the most prominent SWMS platforms that have been used widely by scientists.

Discovery Net
Discovery Net is one of the earliest SWMS and many of its concept have been adopted by other leading SWMS. The main function of Discovery Net is to orchestrate execution of tasks on various remote resources based on the OGSA or Web-service standard. An interesting feature of Discovery Net is its ability to infer dependencies and execute only the subset of a workflow description which is required for producing the requested output.

Kepler
Kepler is a feature-rich SWMS built on top of Ptolemy II coordination engine. Kepler uses dataflow approach and supports Non-DAG workflow description. A unique feature of Kepler is that it separates models of computation from the workflow model. A system called directors allows a workflow to be executed under one of the available MoC, such as the Process Networks, Synchronous Dataflow, Continuous Time, Dynamic Dataflow and Discrete Events.

Kepler uses the MoML-format that is based on XML to store its workflow description. Some of the most notable features includes GUI-based and command line-based workflow composition, user-defined workflow scheduling strategy, and all three (centralized, mediated, peer-to-peer) data movement strategy (as explained in Section 2.3.2). Currently, Kepler is amongst the most feature-rich and extensible open-source SWMS available.

Taverna

Taverna is an SWMS that supports DAG-based workflow description, described using its own XML-based format called SCUFL. In contract to Kepler which supports multitude of MoC, Taverna supports a hybrid model that is a combination of dataflow and functional based on the lambda-calculus theory for what is known as a collection-oriented processing[36].

Triana

Triana supports Non DAG-based workflow description that is described using its own format similar to WSFL[50]. In addition, it also supports workflow description in the WSRF language from OASIS. Triana, as opposed to both Kepler and Taverna supports decentralized workflow scheduling architecture. It uses peer-to-peer as its sole data movement strategy.

BPEL

BPEL is a non-scientific workflow management systems centered around the use of web-services as components. In BPEL, a workflow description is written in the BPEL4WS-format which is based on XML.

BPEL was designed for use in the business domain and have yet to see any major adoption in the scientific domain. In [16] it was argued that this is partly due to the lack of abstraction present in BPEL4WS, as service-composition are being dealt with at the level of web service ports. However, there are some ongoing efforts to adapt BPEL by raising the level of abstraction to suit scientific needs, such as the work on OMII-BPEL or GPEL4SW[46].

Chapter 3

WeevilScout Architecture

This chapter describes the architecture of WeevilScout, our approach to enabling distributed computing on an ensemble of web-browsers. WeevilScout is composed of two major parts, the *WeevilScout server* and thec*remote workers*. Both components will be carefully explained in the next few sections.

3.1 High Level Architecture

WeevilScout is a distributed, web-based scientific workflow management systems (SWMS). The first prototype allowed for the execution of Javascript jobs in remote worker machines[40]. Since then, new features such as a workflow coordination engine and WebCL job type have been developed.

As an SWMS, WeevilScout coordinates the execution jobs using workflows. A *job* is defined as something that produces a *value*. A job is implemented as a Javascript function that can also contain OpenCL *kernels*, in the case of a WebCL job. During its lifetime, a job can receive input and send output to another with the help of the coordination engine that we have developed.

As defined in Section 2.3.1, a *workflow* is a collection of jobs and how the dependencies are structured. WeevilScout uses *dataflow* models of computation to define these dependencies; that is, a job is scheduled for execution if and only if all of its input requirements have been met.

As a web-application, WeevilScout is composed of two major parts, the *WeevilScout server* and *remote workers*. A WeevilScout server consists of *five* major components: a *web-based front end*, a *workflow parser*, a *dataflow scheduler*, a *job-queue*, and last a *performance monitor and visualization*.

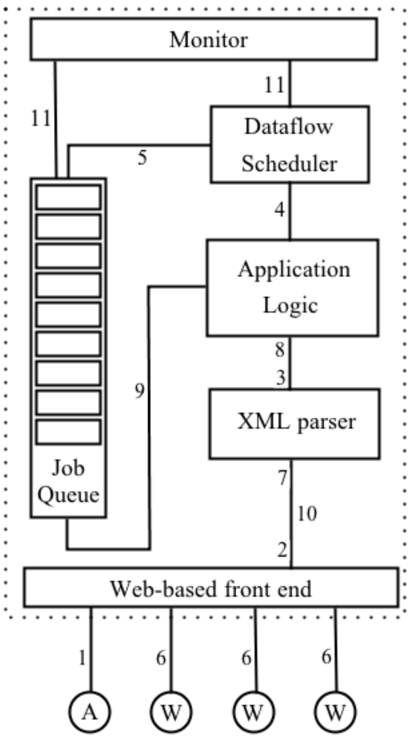

Figure 3.1: Major WeevilScout server components

The main components of the WeevilScout server resides within the dotted box in Figure 3.1. External entities such as the administrator and remote workers are denoted as circles with an A and W, respectively.

The numbers in Figure 3.1 is used to show the typical lifecycle of the system which can be described as the following: (1) an administrator responsible for managing WeevilScout will submit an XML-based workflow description to the WeevilScout server via its web-based front end. (2) The workflow description will be parsed and (3) an actual dataflow graph will be built according to this description. (4) The graph will then be passed down to the scheduler which is responsible for managing the (5) queueing of jobs. (6) One or more remote workers will send periodical request in XML format, asking for available jobs. (7) This message will be parsed and (8) interpreted by the application logic that will contact the job queue. (9) Upon being contacted, the job queue will return an available job if any. (10) The job description will be serialized into XML before being sent to the requesting browser.

For executing jobs, WeevilScout relies on the existence of *remote workers*. Generally, a remote worker is defined as any kind of computing resources with a web-browser application that supports Javascript, WebWorker, and (optionally) WebCL. Some example of such resources are consumer-grade CPUs that might or might not be equipped with a GPU. A mobile device could also be fitted into this

category, although power-consumption issue might prove it as being less ideal for this application.

Characterizing The Environment

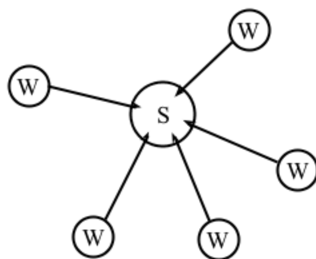

Figure 3.2: Centralized resource gathering and communications

In WeevilScout, communication is centralized (i.e. Figure 3.2) with the WeevilScout server acting as the central hub of communication between remote workers, and computer networks the main channel that connects them. Computer networks are inherently unreliable and often provide limited QoS. Even more, jobs will be delivered to remote workers, which are essentially a collection of anonymous machines on various locations around the globe which cannot be held accountable. It must always be assumed that remote workers are *faulty*, meaning that there is a high chance for them to produce incorrect result. For example, it might happen if a remote worker uses a certain beta version of a web-browser.

Joining WeevilScout is as easy as pointing the browser to the URL of the machine that hosts a WeevilScout server. This is one of the main motivation of WeevilScout, to allow remote workers to easily join in the computation. However, for the very same reason they are also able to leave at any time, simply by closing the browser window. As the time when a worker might join and leave cannot be predicted, it is always encouraged to program jobs with small workload that will not take too much time to complete. The application of highly-granular jobs minimizes the amount of work loss in the event of failure such as a worker that crashes. In addition, it means that more parallelism can be exploited by allowing these jobs to be distributed to more remote workers.

3.2 Work Unit

A work unit is the smallest entity that can be defined. In WeevilScout it is a Javascript function, which from now will be referred to an a *job*. WeevilScout supports two types of job: *WebWorker job* and *WebCL job*. A WebWorker job is a

normal Javascript program that is executed in the background with a WebWorker instance. A WebCL job is written in both OpenCL and Javascript, with the later acting as the coordinator of the OpenCL kernels.

A special function `weevil_main` must always exist in any job description and serves as the entry point of execution. `weevil_main` can be defined with an arbitrary number of parameters.

3.2.1 WebWorker Job Description

```
1 function weevil_main(a, b) {
2     return a+b;
3 }
```

Figure 3.3: A WebWorker job Description

A WebWorker job consists of one or more Javascript functions, and each can contain an arbitrary number of parameters. While all Javascript language constructs are supported, a few built-in Javascript objects are not available. For example, the Document Object Model (DOM) and most properties in the `window` object of the WeevilScout page are not available.

Source Code Transformation

A necessary transformation step will be performed to transform the code representation (see fig. 3.3) into a representation which can be properly accepted by the `Worker` constructor.

```
self.addEventListener('message', function(e) {
    var data = e.data;
    switch (data.cmd) {
        case 'start':
            weevil_main();
            break;
        case 'stop':
            self.close();
            break;
    }
    function weevil_main(){
        var a = e.data.a;var b = e.data.b;
        self.postMessage( a+b);
    }
}, false);
```

Figure 3.4: The code from Figure 3.3, after transformation steps

Figure 3.4 shows the typical result of applying code transformation. Transformation steps are as follows: It can be seen in lines 11-14 that the original function is kept with some modifications, such as the removal of function parameters. Parameters are now received via *message*, contained within the `e.data` object. `return` statement in the original function are replaced by a `postMessage` invocation, which instead sends a message to the parent thread i.e. the WeevilScout main page.

After performing the steps above, the modified function will be wrapped within an anonymous function (see line 1). This anonymous function is registered as an *event listener* that will be triggered whenever there is a `postMessage` invocation from the WeevilScout main page. Aside from parameter data to be passed to the original function, the object `e` also contained a parameter `cmd`, which is used to start or stop the worker. The main page will tell the worker to stop when it has received the result.

3.2.2 WebCL Job

WebCL jobs (see Appendix for an example) are different than WebWorker jobs, as they have direct access to the hardware. A WebCL job is designed to harness the power of OpenCL compute devices available in remote workers, such as a multi-core CPU or GPU to allow for a faster and more efficient execution.

A WebCL job consists of one or more OpenCL *kernels*. A kernel is a unit of OpenCL code that will be distributed and executed in one or more available OpenCL compute devices. A certain function within the WebCL API called `createProgramWithSource` takes a string of OpenCL code that will be compiled into a kernel object that can be immediately executed. Another function `setKernelArg` is used to set various arguments of the kernel. Finally, `enqueueNDRangeKernel` will enqueue the kernel object for execution. The function takes a number of arguments such as pointer to the kernel object, local and global work unit, and ND-range. An ND-range parameter is specified to tell OpenCL whether to treat the data as a 1, 2, or 3 dimensional data mainly for convenience such as when working with images or 3D meshes.

It is important to mention that the WebCL API provides the sole interface between the OpenCL and Javascript environment. The programming model dictates that these two environments are separate; Javascript knows nothing about the kernel code and does no attempt to check the validity of input on either syntactic or semantic level. On the positive side, as these environment are separate, a crash in an OpenCL kernel, such as when accessing bad memory address, will not propagate to the browser environment.

3.3 Workflow

A single job is very limited in its usefulness. In order to allow coordination of multiple job instances, a coordination scheme must be devised. In WeevilScout, this is achieved via workflow mechanism which is described by using our own XML-based workflow description format. It supports the *static dataflow* MoC that is realized by sending a job for execution only when all of its input requirements are available.

After considering both Dataflow and Process Networks, the two models of computation we feel most suitable for WeevilScout, there are two reasons of choosing the former. The main reason is when the environment of which WeevilScout is designed for - the internet - is considered. As it is explained in Section 3.1, the environment consists of *anonymous* workers connected through a possibly *faulty* computer network, and thus unreliable in general. To mitigate the risk of losing work due to various conditions, jobs must be made as short as possible. This rules out Process Networks MoC which are better suited for job that is designed to run asynchronously for a long time, for example. The other reason is that dataflow concept is very intuitive and easy to understand, resulting in a shallow learning curve.

A workflow is described in terms of the set of nodes and edges it contains. The description follows our own XML-based *workflow description format*, as will be described in Section 3.4.

When a workflow is executed, a new *instance* of that particular workflow will be created, implying that it is possible to spawn multiple instances of a certain workflow description. This could be useful for running an experiment multiple times using different data sets, for example.

3.3.1 Workflow Structure

WeevilScout implements a DAG-based workflow structure. Each *node* (or vertex, in graph term) could represent a WebWorker or WebCL job instance, or a simple *value* amongst others that will be described in more details in Section 3.3.2. *edge* represent dependencies between nodes i.e. which output from a node goes to other nodes. Control flow features such as loops and conditionals are not implemented to stay true to the spirit of dataflow execution model.

A workflow can start with any number of *starting* nodes, but must have exactly one *final* node. A final node will represent the final result of a particular workflow instance based on same particular input.

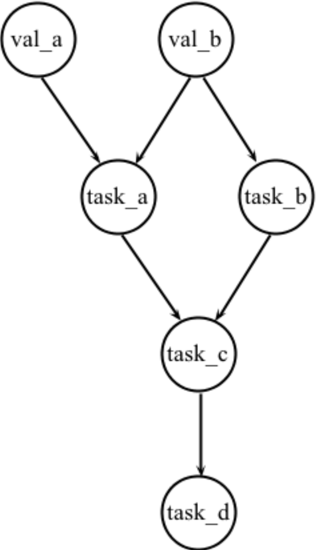

Figure 3.5: A workflow description

Figure 3.5 shows a simple workflow description consisting of two *value* nodes, `val_a` and `val_b`, and four job nodes, `task_a` through `task_d`. A node represents an *instance* of a certain value or job. This implies that there can be multiple instances of an identical value or job description. For example, both `task_a` and

`task_c` can be an instance of a job description `add.js`, while `task_b` and `task_d` from `roundValue.js` and `convertToString.js`, respectively. `val_a` and `val_b` are the *starting nodes* of that particular workflow description, as they are the only nodes which does not have incoming edge(s) and thus can be scheduled for execution right from the start. Finally, `task_d` is the *final* node.

Edges

An *edge* implies an *input-output dependency* between two nodes, that is, which output from a certain node will be sent to which other node as an input. For example, in Figure 3.5, there is a such dependency between `val_a` and `task_a`, `val_b` and `task_a`, `val_b` and `task_b`, and so on.

Ports

Ports allow for incoming data from other nodes to become arguments of the `weevil_main` function. For that to happen, a port must be *named* with the same name as the parameter name of the function. A port is in the `OPEN` state if it is still waiting for input. Upon receiving the necessary input data, state will be updated to `CLOSED`.

3.3.2 Workflow Model

Nodes in WeevilScout borrow the concept of *functions* in most programming languages: just as a function that is implemented as an entity that can receive multiple arguments and return a single value, a node can have an arbitrary number of input ports (or none at all), but *exactly* one output port. This model is chosen since jobs, which are arguably the most important elements in a workflow, are basically Javascript functions which only allows exactly one return value.

However, having only one output port does not imply that a node can only have one outgoing edge. On the contrary, it is possible to have multiple outgoing edges to send a result to multiple nodes (as in Figure 3.5).

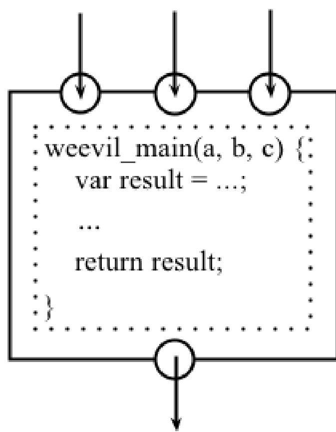

Figure 3.6: Representing a job as a node within a worklow

To motivate the readers with an example, Figure 3.6 shows a node containing a WebWorker job description with three parameters contained within a node instance. Each parameter is represented as a port in the node that contains it (shown by the small circles). Therefore, the node contains three input ports and one output port.

As briefly mentioned previously, a node can also represent a number of other things besides a single WebWorker or WebCL job description:

Simple Nodes

Value node

A *value node* is the simplest kind of node: it simply contains a value. A value can be of any valid Javascript data types, which are `Number`, `String`, `Boolean`, `Object`, or `Array`, `null`, and `undefined`. Value nodes are often the first nodes to be defined in a workflow, to act as initial input to other nodes (such as in Figure 3.5).

In a workflow instance, a value node is not special, that is, there are no particularly interesting steps done to accommodate it, such as a special scheduling steps, conversion, *et cetera*. When a job that depends on a particular value is scheduled for execution, that value will simply be retrieved and sent to the input port of that job node.

Job node

A *job node* consists of either a WebWorker or WebCL job description. The node will contain the same number of ports as the number of parameters that the job description accepts. Moreover, a job node will always have an output port, just as how the job description will always have a return value (which is `undefined` if it does not contain any `return` statement.

Upon encountering a job node during a particular workflow execution, the scheduler will check if all of its input ports already contain the data they need. A node will be put into the *job-queue* if all of its input ports are in the `CLOSED` state (they have received the required data). It goes without saying that if all ports in a particular job node happen to be the receiving end of some value nodes, execution will begin almost immediately, since the required data are instantly available.

Complex / Meta Nodes

Pool collection node

A pool collection is designed to model the SIMD[17] concept. It takes a set of values and distributes each of them into an array of processors. Each processor instances contains identical code and will allow parallel processing of value. Finally, the results are aggregated in a reducer node where reduction operation can be performed. The reducer is guaranteed to execute only when all of the processors have finished executing.

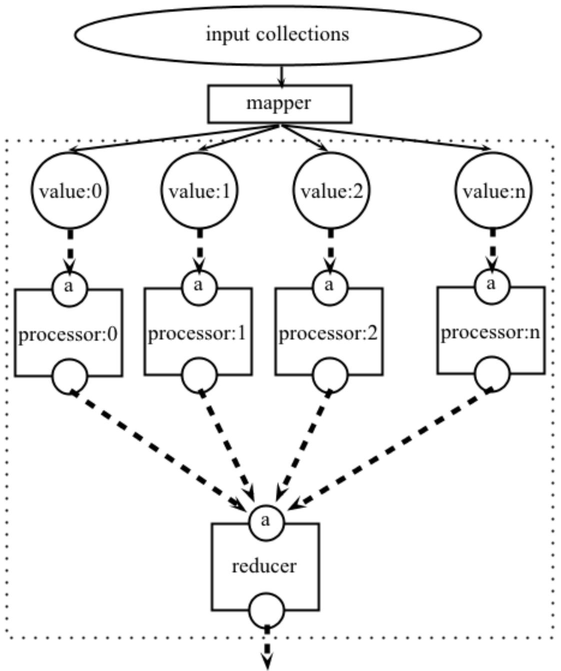

Figure 3.7: Structure of a pool collection

The actual structure of the pool collection node can be seen in Figure 3.7. A special function called *mapper* of the pool collection will map each element in the input collection into an instance of a *processor*.

Both *processor* and *reducer* are a specialization of the job node, in that they can only have one input port. Moreover, the reducer's port can only accept a single data of `Array` type. This restriction is by design, with the reason that a reducer will need to receive input from an array of processor nodes.

Figure 3.7 shows the structure of a pool collection node, which consists of the following elements:

1. A set of n value node: Each of these value nodes represents an element from the array input and serves the purpose of passing each element into the subsequent job node.

2. A set of n job node: A job node receives and process input coming from its associated value node. This is not a regular job node, but rather, a specialization version as it only allows *exactly* one input port.

3. A reducer node: A reducer node is also a specialization of the regular job node where it only allows for an array parameter input. This parameter will be filled in when all jobs have finished executing and is meant for a reduce operation (which is a common operation in parallel programming).

4. A container node: A container node is the outer node and does the communication with other nodes in a workflow description. These nodes never knows about the existence of internal value, job and reducer nodes that exist within the pool collection node.

3.4 Workflow Description Language

A workflow is described in our own XML-based, simple workflow description language. We looked at common graph description language such as GraphML[11], but we feel that it would be too complex and require too much time to integrate it into our framework. The Scala language provides an extensive support for XML processing, which makes it easy for us to create a simple parser to support our own description language for WeevilScout.

The workflow description will be stored in a plain-text file i.e. `Workflow1.xml`, referred to as the *workflow document*. An administrator can choose to execute a workflow from the workflow document that must be accessible by the WeevilScout server, or by typing the workflow description directly into the parser from the web-based front end.

3.4.1 Elements

As an XML document, the workflow document contains tags that is written and structured according to the typical XML convention. For example, a tag can be enclosed within another tag. Also, a tag can have multiple attributes to alter its properties.

Structure

A proper workflow document must be structured according to the following structure:

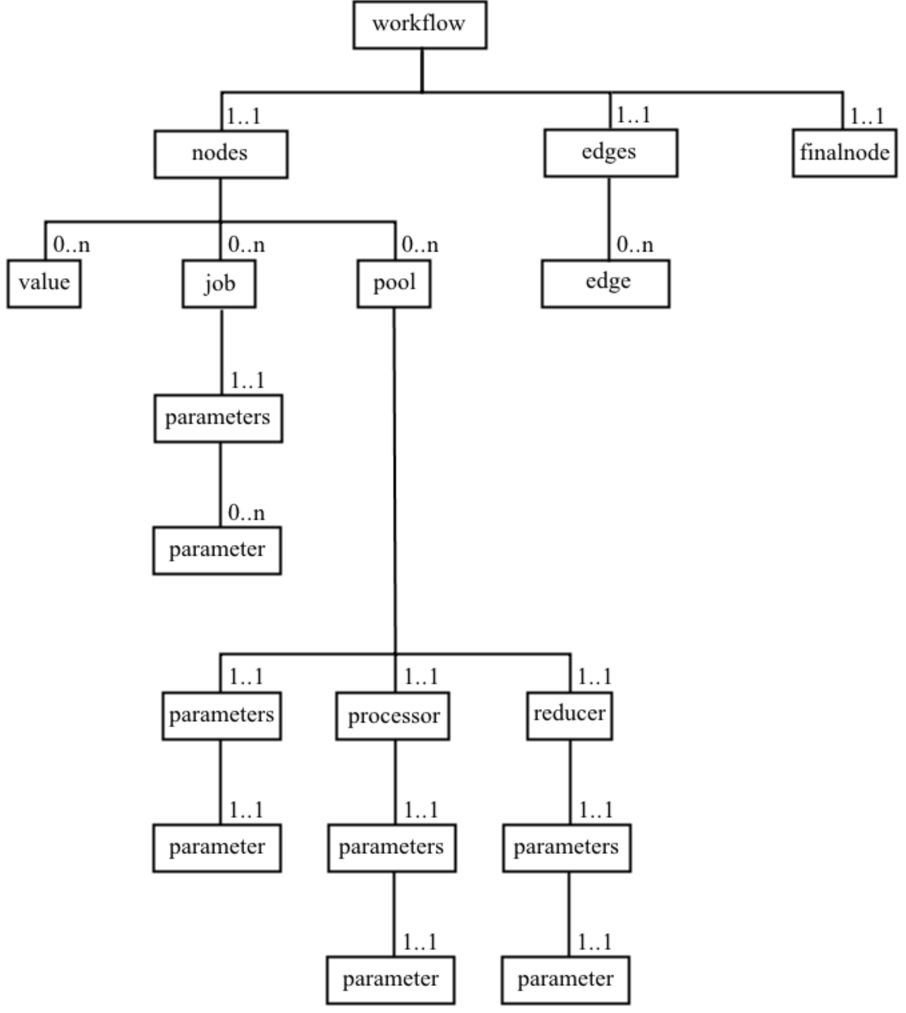

Figure 3.8: Tree Structure of The Workflow Description Language

In this diagram, the box represents an *element*, also known as *tag*. An element can contain other elements: in the diagram this relation is described by a line that connects two elements, e.g. `final` must be nested within `workflow`, `parameter` within `parameters`, and so on.

Another property of a relation is its *cardinality*. A *1..1* cardinality indicates that a particular element can only occur once within its direct parent, e.g., there can only be one occurrence of `parameters` within a `job`. In contrast, a *0..n* cardinality indicates that there can be an arbitrary number (or none at all) of elements within its direct parent, e.g., there can be as many `parameter` occurrence within a `parameters`.

Element Descriptions

This section contains the list of all elements that are allowed within a workflow document. Each element can have various *attributes*, as will be describe shortly.

workflow

The `workflow` tag must only occur once and be the root element in a workflow document. It marks the start and end of a workflow description. The `workflow` tag does not have any attribute.

nodes

The `nodes` tag must only occur once within a `workflow` tag. It encloses the list of nodes in a workflow. The `nodes` tag does not have any attribute.

edges

The `edges` tag must only occur once within a `workflow` tag. It encloses the list of edges in a workflow. The `edges` tag does not have any attribute.

finalnode

The `finalnode` tag must only occur once within a `workflow` tag. It contains a single attribute, with values that must match an id of an element to mark it as the final node. The element it points to must be an element of type `value`, `job`, or `pool`, i.e., `processor` or `reducer` is not allowed.

Attribute	Description
`id`	Must contain the id of the node that will be destined as final

value

The `value` tag is the direct representation of the *value node*. It contains a single value, and can have the following attributes:

Attribute	Description
`id`	Identifier of the node
`value`	Value that the node will contain (is mutually exclusive with `src`)
`src`	Loads a value from the specified source file (is mutually exclusive with `value`)
`returntype`	The data type of the value it holds

job

The `job` tag is the direct representation of the *job node*. It contains a job description, and can have the following attributes:

Attribute	Description
`id`	Identifier of the node
`jobtype`	Must be set to either `javascript` (a WebWorker job) or `webcl` (a WebCL job)
`src`	Loads the job description from the specified path
`returntype`	The data type of the value that will be returned by the job description

pool

The `pool` tag is the direct representation of the *pool node*. It contains the *processor node* and *reducer node*, and can have the following attributes:

Attribute	Description
`id`	Identifier of the node
`returntype`	The data type of the value it holds

processor

The `processor` tag is the direct representation of the *processor node* that can only exist within a pool node. It contains a processor description and can have the following attributes:

Attribute	Description
`id`	Identifier of the node
`jobtype`	Must be set to either `javascript` (a WebWorker job) or `webcl` (a WebCL job)
`src`	Loads the job description from the specified path
`returntype`	The data type of the value that will be returned by the processor description

To comply with the restriction defined in Section 3.3.2, a check will be performed

to ensure that there is only one instance of `parameter` within the `parameters` of a particular `processor`.

reducer

The `reducer` tag is the direct representation of the *reducer node* that can only exist within a pool node. It contains a reducer description and can have the following attributes:

Attribute	Description
`id`	Identifier of the node
`jobtype`	Must be set to either `javascript` (a WebWorker job) or `webcl` (a WebCL job)
`src`	Loads the job description from the specified path
`returntype`	The data type of the value that will be returned by the reducer description

To comply with the restriction defined in Section 3.3.2, a check will be performed to ensure that there is only one instance of `parameter` within the `parameters` of a particular `reducer`. Moreover, that sole instance of `parameter` must have its `paramtype` attribute set to `Array`.

edge

The `edge` tag represents a connection between a port in two different nodes, i.e. their dependencies. It can have the following attributes:

Attribute	Description
`id`	Identifier of the edge
`source`	The id of the node that acts as the sender of result
`dest`	The id of the node that acts as the receiver of result
`paramdest`	Port destination of the receiving node

parameters

The `parameters` tag encloses a list of parameter that a `job`, `reducer`, `processr`, or `pool` accepts. The `parameters` tag does not have any attribute.

parameter

The `parameter` tag represents a port that belongs to a node. It can have the following attributes:

Attribute	Description
`id`	Identifier of the port
`paramtype`	Datatype that is accepted by this port

3.4.2 Parsing Steps

When a workflow description is received from a workflow document or string input from the administrator, it must be parsed; a step that consist of converting textual descriptions into a *DAG object* that can be understood by the dataflow scheduler. At the most abstract level, the DAG object is constructed from the elements defined in the workflow description. After the DAG object has been constructed, it will further be augmented with various informations that are needed for execution. As will be described shortly, the process of fully constructing a workflow instance follows these four steps.

Building a DAG Object

The XML-parser will collect all elements within the `nodes` and `edges` element, i.e., `value`, `job` and `edge`, with the following algorithm.

```
1  Set[Element] nodeset = getXmlElements("workflow \\ nodes \ _")
2  Set[Element] edgeset = getXmlElements("workflow \\ edges \ _")
3  Node finalnode = getXmlElements("workflow \ finalnodes")
4  Graph g = build(nodeset, edgeset)
5  g.setFinalNode(finalNode)
```

Figure 3.9: Parsing a workflow description

Scala language features an easy to use XML selector mechanism that can be used to retrieve elements and attributes easily, and `getElements` was implemented based on it. Furthermore, the Graph4Scala[1] library was used as the basis for our DAG data structure.

The `build` function takes the set of nodes and edges to construct a DAG object.

```
1  def build(Set[Element] nodes, Set[Element] edges):
2      Graph g
3      foreach n in nodes:
4          Vertex v = createVertex(n)
5          g.addVertex(v)
6      foreach e in edges:
7          Vertex src = find(e.source.id, g.vertices)
8          Vertex dst = find(e.dest.id, g.vertices)
9          Edge edge = createEdge(src, dst)
10         g.addEdge(edge)
11     return g
```

[1] https://www.assembla.com/spaces/scala-graph/documents

After executing the steps in Figure 3.4.2, we will end up with a DAG object `g` where its vertices represent nodes in a workflow instance, e.g, a job, value, pool node, *et cetera* and its edges the dependencies between the nodes.

Preparing For Execution

Type Checking

We feel that enforcing types on both input and output ports of a node is beneficial in our workflow system. A type-checking mechanism will check if an input data of a particular type will be sent to an input port with compatible type. Sending data from a `String` port to a `Number` port would fail, for example.

Our simple type-checking mechanism is achieved with the existence of the `paramtype` and `returntype` attributes that must be present for all nodes and ports definition.

```
1        <job id="add0" src="Add" returntype="Number">
2            <parameters>
3                <parameter paramtype="Number" id="a" />
4                <parameter paramtype="Number" id="b" />
5            </parameters>
6        </job>
```

Figure 3.10: The `paramtype` and `returntype` attribute in action

Figure 3.10 shows how to apply the type checking attributes to a job description. The `paramtype` attribute must be attached to each port definition, i.e, the `parameter` tag. It defines what kind of data should be expected by that particular input port. On the other hand, the `returntype` attribute defines the type of the output port and should be attached to the `job` element in the description.

Since both WebWorker and WebCL jobs use Javascript as the interfacing language[2], all valid Javascript data types are allowed as port types. The valid Javascript data types are `Number`, `String`, `Array`, `Object`, `undefined`, and `null`.

Preventing Race Conditions

In most workflow structures, there will come a situation where there exists a vertex with multiple incoming edges from two or more vertices. Conceptually, this indicates a node with two or more incoming ports that will accept data from other

[2]See Section 3.2

nodes. However, dataflow scheduler executes each node concurrently[3], causing race conditions should two or more nodes writes to the same vertex.

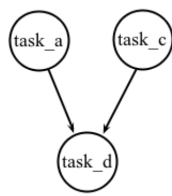

Figure 3.11: A vertex with two incoming edges

In the face of such a problematic and potentially dangerous situation, we augmented each node with a copy of a partial information of the destination node, preventing race conditions from happening since each node will only write to its own local copy of data.

```
1 function visit(Vertex v) {
2     Node n = v.getAssociatedNode()
3     foreach e in v.getEdges():
4         Vertex destVertex = e.dest
5         n.addDestinationNode(destVertex.getAssociatedNode())
6 }
7 g = DepthFirstTraversal(visit)
```

Figure 3.12: Copying informations within a vertex to its associated node

Figure 3.12 shows a depth-first traversal done to a DAG object, augmenting its associated node with information about all of its outgoing vertices.

Assigning Supervisor Nodes

A *supervisor* node is the node responsible for starting up its children[4]. In the case of a node with only one parent node, making the decision is trivial. However, in a more complex case such as in Figure 3.14, a question is raised whether it is `task_a` or `task_b` that should start `task_d`.

Our solution to this problem is to pre-assign the supervisor node for each children and is done before the execution begins by examining the DAG object. In case where there are more than one potential supervisor candidate for a particular child, as in Figure 3.14, the node that appears first within the list of parents will be appointed as its supervisor.

[3]Described in Section 3.6.2

[4]In practice, this is the job of the executor associated with the said node, as will be described in Section 3.6.2

Calculating Node Paths

Furthermore, each node is augmented with its *path*. A path is a pre-assigned string stored in each node that consists of the node ID, its parents up to the root node IDs, the workflow ID, and finally a predefined sequence of strings (according to Scala's actor path convention) in a hierarchical fashion that follows the structure of the DAG object. This path is used to refer to a node without having to do a search on the DAG object, particularly useful when passing down results from remote workers to the appropriate node.

For example, the path to `task_d` in Figure 3.14 will be the string `/user/rootSupervisor/<workflow_id>/task_a/task_d` where `<workflow_id>` refers to the pre-assigned ID of a specific workflow instance.

The following algorithm traverses and pre-calculates node path for all nodes in a DAG object.

```
1  precalculateNodePath(node):
2      if (!node.traversed):
3          node.traversed = true
4          node.path = node.id + "/"
5          for (s in node.successors):
6              if (s in node.assignedSupervisorCollection):
7                  s.path += node.path.clone()
8                  precalculateNodePath(s)
9
10 rootNodePath = ["/", "user/", "rootSupervisor/", workflowId+"/"]
11 for (n in startNodeCollection):
12     n.actorPath = rootActorPath.clone()
13     precalculateActorPath(n)
```

Figure 3.13: Algorithm to pre-calculate node path

The algorithm works by doing a recursive depth-first traversal into a DAG object. Each node contains a flag that indicates if it has been traversed, and if it has, the algorithm can skip it. Upon visiting a node, it will append the ID of that particular node into the path. In addition, as shown in the algorithm, a path must always start with the string `/user/rootSupervisor/<workflow_id>`, where again, `<workflow_id>` refers to the pre-assigned ID of a specific workflow instance.

A Properly-Augmented DAG object

Figure 3.14 shows a properly-augmented DAG object that results from executing the steps as enlisted above. For the purpose of demonstration, it is assumed that the workflow is pre-assigned an ID `wf0`.

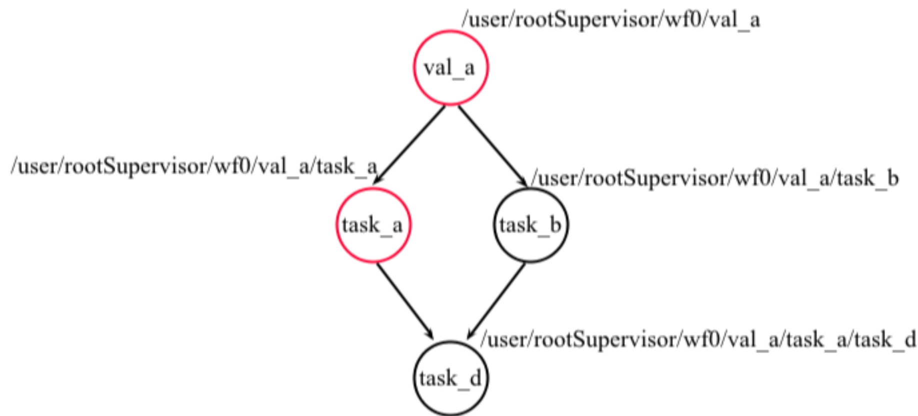

Figure 3.14: A properly augmented DAG object

In Figure 3.14, the structure contains four nodes that are connected by some edges. The red circle indicates a supervisor node and the black circle a normal node. Each node stores its own node path, which will be used by the web-based front end component to identify the right node to send the result to. Although not shown, each node contains partial information about its direct children, e.g., `val_a` stores partial information of `task_a`, and `task_b`, and so on.

3.5 Scheduling and Execution

In the previous section we have described components of a workflow. In order for these workflow components to execute in a coordinated, meaningful way, there must be a coordinator of some sort to manage them. This is the job of our *dataflow scheduler*. The scheduler is at the core of our dataflow-based workflow management and does various important things: it decides when to mark a particular node for execution, where to send result data based on a particular workflow structure, retrying a failed request, and other things that will shortly be described in depth.

During the project, we have tried various approach of modeling dataflow execution, and we have deemed the actor model to be more preferable than others as they offered the right level of abstraction - it would be a waste of time and effort to start writing dataflow scheduler logic using low-level approaches such as

threads and locks, for example. Neverthless, some of the approach we tried have also been documented here.

3.5.1 Alternatives

This section contains two approaches that we have considered before we decided on the implementation using the actor model.

BFS-Based Traversal With Traceback

When we first consider the problem at hand we viewed the problem as a graph problem that can be solved using combinations or variations of traversal algorithm. The BFS-based traversal with traceback is exactly so - it does a graph traversal similar to a BFS algorithm, and performs a traceback to start traversing into a different node when an execution cannot continue from a certain node.

We start with an empty queue `q` and fill it with a set of start nodes, and as we visit each node we add all outgoing nodes in the queue. The pseudo-code is as follows:

```
1  w = parseWorkflowDescription()
2  q = new Queue()
3  for each node in w.getStartNodes():
4      q.add(node)
5
6  for each node in q:
7      visit(node)
8      for each outnode in node.getNodeFromOutgoingEdges():
9          q.add(outnode)
10
11 function visit(n):
12     scheduleForExecution(n)
```

Figure 3.15: Algorithm for the breadth-first execution

Figure 3.15 shows how the algorithm of breadth-first execution is implemented. Since traversal is done in a breadth-first fashion, the order of elements that will be inserted into the queue guarantees that the order of execution is preserved.

However, deadlock can occur when executing a workflow that follows a specific structure as described below:

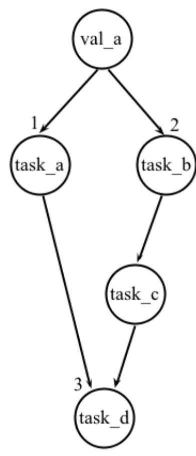

Figure 3.16: Structure that leads to deadlock during an execution

The numbers in Figure 3.16, shows the order of element being traversed. Specifically, when the point of execution reaches `task_d`, the algorithm encounters a deadlock since it needs the input from `task_c`, which are not scheduled for execution.

To prevent this particular situation, a traceback mechanism can be introduced. The pseudo code of the algorithm is as follows:

```
1 function visit(n):
2     foreach p in parent(n):
3         if (p is not scheduled):
4             visit(p):
5         else:
6             scheduleForExecution(n)
```

Figure 3.17: Adding traceback behavior in function `visit`

Function `visit` is modified with the following behavior: when it visits a certain node, it will not immediately schedule that particular node for execution. Instead, first check if one or more of its parents have all been scheduled. If not, visit that parent instead. Visiting a parent will perform the same check on its parent (which is the grandparent of the original node), and the traceback step will continue until it encounters a node of which all of its parents have been marked for execution. When the algorithm finishes, it will find an order of execution that will be safe from deadlocks.

We decided that this algorithm could work, but will require a significant effort to implement even for the simplest case. Moreover, handling unexpected conditions that could happen during execution (e.g. rescheduling a node whose job is

being handled by a worker that crashed) will require additions that would significantly complicate the basic logic. The main reason is because this algorithm only works at the *static* level; it does not take into consideration any dynamic information from runtime. For example, it will be hard to schedule a pool collection node properly since the number of processors that will be scheduled depends on the number of elements in the input.

Threaded BFS Based Traversal

Traceback is necessary to get the algorithm to work but introduces unwanted complication to the otherwise simple algorithm to accommodate for the surprisingly common pitfalls. As we have observed, the only case where traceback will be needed is when there a diamond-shaped structure where one of it's *side* have more nodes than the other:

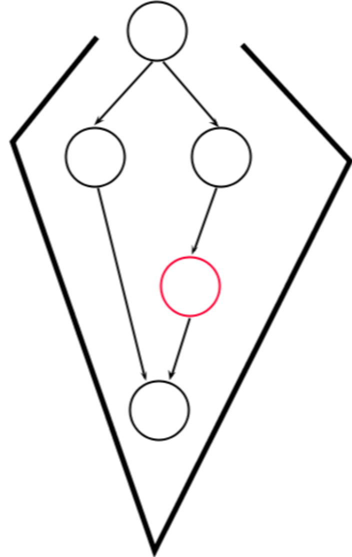

Figure 3.18: The diamond-shaped structure with uneven levels

In Figure 3.18, the right side of the diamond structure contains 1 more node than the left side (indicated by the red circle), which causes deadlock when a traceback step is not introduced. An alternative to the traceback mechanism is to introduce concurrent traversal mechanism. Since the structure is traversed from different *directions*, a traversal that reach a dead-end can wait for the other traversals to finish, fulfilling the unmet dependencies that causes the first traversal to deadlock. The main problem is determining how many concurrent traversal instances n are needed. Based on our experiment, we have observed that the amount will need to be at least equal to the number of incoming edges at the

merging node, (indicated with an M in Figure 3.18). Since it has two incoming edges, $n \geq 2$.

This approach provides a different alternative for solving the deadlock problem. However it will still introduce significant complexity when the logic for handling unexpected conditions during runtime is to be introduced.

3.6 Dataflow Scheduling Using Actor Model

After considering the other alternatives, we decided that scheduling should be implemented using the actor model.

This section will describe how our dataflow scheduling mechanism is simply made up of a collection of actors. By modeling all kinds of entities, e.g., jobs, values, workflows, as an independent, communicating agent we were able to achieve the goal of a fully functional dataflow-style execution while maintaining the simplicity of design.

3.6.1 Root Supervisor

A root supervisor sits at the top-most level of the actor hierarchy. Its primary role within the scheduler is to manage a collection of workflows by actions such as spawning a new instance and terminating those that have finished.

3.6.2 Executors

During an execution of a particular workflow, each node will be associated with an *actor* instance. Actors, as described in Section 2.1.1, are concurrent agents that can communicate with other actors. For the context of the paper, these actor instances are referred to as *executors*. An executor keeps track of the state of execution of the node associated with it. If a node have received all the required input, the associated executor will schedule it for execution. If it has not, the executor waits for messages from the other executors that contains the required input. Just as importantly, executors are the entities that send and receive input data on-behalf of the node associated with it. An executor takes care of setting the port state of a node to `CLOSED` when it receives an input message for one of the port.

Just as there are many types of nodes e.g. value, job and pool nodes, there are also many types of executors: a value executor, job executor and pool executor.

Different executor types behave differently and will be elaborated in more details in Section 3.6.4.

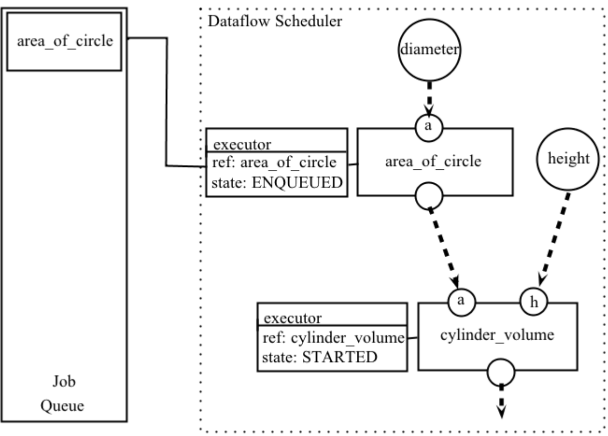

Figure 3.19: Two job executors (value executors not shown) in different states

Figure 3.19 shows how a workflow instance is being executed within WeevilScout dataflow scheduler. A circle indicates a value node while a box indicates a job node. Each node is associated with a single instance of executor. Every time a job executor receives new input, it send them to the node and set the receiving port state to `CLOSED`.

Executors have internal states, are concurrent, and communicate with other executors via message-passing. When an executor receives a message, it may send any data that is contained to the node it is associated with. In addition, it may also change its *internal state* accordingly, depending on the message content.

3.6.3 Messages

A message is an immutable object containing one or more values that are assigned during object creation. Executors communicate with each other by sending and receiving messages, i.e., they never modify the internal states of others directly and nor they have to. This makes it easy to understand the emergent behavior that results from communication between executors. Moreover, dealing with immutable objects reduces the likeliness of race conditions. The design principle of actor models encourages side-effect free communications, limiting operations with side-effect within an actor instances boundary. We have adopted this model for programming our executors to ensure robustness in their behavior.

Sending a message from an executor to another can either be a *blocking* or *non-blocking* operation. The former implies synchronous execution, waiting for the other end to have received the message before execution in the sender can

continue. The later implies asynchronous execution, where the sender does not wait for the message to arrive before continuing.

3.6.4 Executor Types

Workflow Executor

Upon receiving a DAG object, the root supervisor will spawn an associated *workflow executor*. Termination happens when the final node has finished executing. The workflow executor has the following state transitions that define the behavior:

1. `STARTED` An executor will reach the `STARTED` state when it has just been started by the root supervisor. Afterwards it will start all executors of nodes that do not have any input port, i.e., the starting nodes.

2. `FINISHED` After the final node receives its result, its executor will send a notification telling the workflow executor to transition to the `FINISHED` state, does a clean-up procedure, and terminates after notifying the root supervisor.

Value Executor

A *value executor* is always associated with a value node. It can be spawned by either the workflow executor (if it is one of the starting nodes) or by another executor, e.g., a job or value executor. The value executor has the following state transitions that define the behavior:

1. `STARTED` A value executor will reach the `STARTED` state when it has just been started by another executor. First, it will start and send the value of its associated value node to all executor of the node that it supervises[5], if any. Then it waits for value request message coming from its non-supervised children, if any. Upon receiving such message, it will send the value of its associated value node as a response until all of its children have received the result. If all of its children have received the value of its associated value node, or if it does not have any non-supervised children, it will transition to the `FINISHED` state.

[5]See Section 3.4.2 about assigning supervisor node

2. FINISHED The value executor will perform a clean-up procedure, notify the root supervisor that it has finished, and terminates shortly.

Job Executor

A *job executor* is always associated with either a job or a processor node. The job executor has, together with the reducer executor, the most complex state transitions amongst all executors, as will be described below.

1. STARTED

   A job executor is in the STARTED state when it has been started successfully. From this state it can transition into the PARAM_NOT_FULFILLED state if there are one or more ports that are still OPEN. If the node that it is associated with does not define any ports, it will go directly to the PARAM_FULFILLED state.

2. PARAM_NOT_FULFILLED

   A job executor is in the PARAM_NOT_FULFILLED state if the associated node still have one or more OPEN ports. Every time it receives a message that contains data belonging to one of these ports, it will set the receiving port state to CLOSED. Then it checks if there are still one or more OPEN port, and if there aren't it will transition into PARAM_FULFILLED.

3. PARAM_FULFILLED

   When a job executor reach the PARAM_FULFILLED state, an immediate transition to the ENQUEUED message will be performed. This state is provided for performing various checks before the actual enqueueing process can begin, such as checking for data correctness or monitoring executor performance.

4. ENQUEUED

   A job executor is in ENQUEUED state when the associated node has been scheduled for execution. When the job is actually executed by a remote worker, the state will be updated to RUNNING.

5. RUNNING

   A RUNNING state indicates that a job is being processed at a remote worker.

6. FINISHED

Upon receiving a result from a remote worker, the Web-based front end will locate to which executor it should send the result to by looking at the workflow and node ID that is sent along with the result. The Web-based front-end will wrap the result into a message and send it to the appropriate executor, where the said executor, upon receiving it, will update its state to `FINISHED`.

A job executor in `FINISHED` state will terminate shortly after notifying the root supervisor and performing the necessary clean-up procedure.

Pool Executor

A *pool executor* is always associated with a pool node. A pool executor does not contain a job to execute on its own, but rather acts as the container of a collection of processor nodes, along with a reducer node. It has the following state transitions that define the behavior:

1. `STARTED`

   A pool executor is in `STARTED` state when it has been started successfully. Since the associated pool node always has exactly one open port, it will immediately transition into the `PARAM_NOT_FULFILLED` state.

2. `PARAM_NOT_FULFILLED`

   A pool executor is in this state if the only port that belongs to the associated node is still in the `OPEN` state. After an input collection is received, it will set the receiving port state to `CLOSED` and transition into `PARAM_FULFILLED`.

3. `PARAM_FULFILLED`

   In this state, the pool executor will divide the input collection it has received into individual elements and spawns as many value nodes as the number of elements within the collection. Each of this value node will be pre-assigned a processor node as its only child. Out of these processor nodes, a single node will be chosen as supervisor of the reducer node, which acts as the destination of results from them.

   Immediately after the creation of all of the value, processor, and reducer nodes, the pool collection will start a value executor for each value node that is present. These executors, in turn, will trigger the creation of processor and reducer executors associated with the processor and reducer nodes.

The reducer node is treated like the final node of a workflow: it will notify the pool executor if it has finished executing so that the pool executor can transition into the `FINISHED` state.

4. `FINISHED`

   A pool executor in `FINISHED` state will shortly terminate after notifying the root supervisor and performing the necessary clean-up procedure.

Reducer Executor

A reducer executor within a pool node is only started when all processor nodes have finished executing. This means that input data will always be immediately available and as such will directly transition to the `PARAM_FULFILLED state`.

1. `PARAM_FULFILLED`

   When a reducer executor reach this state, an immediate transition to the `ENQUEUED` message will be performed. This state is provided for performing various checks before the actual enqueueing process can begin, such as checking for data correctness or monitoring executor performance.

2. `ENQUEUED`

   A reducer executor is in `ENQUEUED` state when the associated node has been scheduled for execution. When the job is actually executed by a remote worker, the state will be updated to `RUNNING`.

3. `RUNNING`

   A `RUNNING` state indicates that a job is being processed at a remote worker.

4. `FINISHED`

   Upon receiving a result from a remote worker, the Web-based front end will locate to which executor it should send the result to by looking at the workflow and node ID that is sent along with the result. The Web-based front-end will wrap the result into a message and send it to the appropriate executor, where the said executor, upon receiving it, will update its state to `FINISHED`.

   A reducer executor in `FINISHED` state will termination shortly after notifying the root supervisor and performing the necessary clean-up procedure.

3.7 Job Queueing

If all ports belonging to a job, processor, or reducer node are in `CLOSED` state, meaning that have received the required input data, the scheduler will insert a new *record* containing that particular job into the *job queue*, waiting for execution in a remote worker. WeevilScout uses a single job queue that is shared between all workflow that are being executed.

The most important operations in the job queue are *enqueue* and *dequeue*. An enqueue operation inserts a new record and is done when, as mentioned above, a node is ready for execution. A dequeue operation takes the oldest record from the queue, which happens when a remote worker contacts the job queue (via the web-based front end) requesting for a job to execute. When this happens, we use the term that a remote worker *dequeues* a job.

3.7.1 Implementation

A job queue can be implemented using various means as long as it displays queue-like behavior. In our case, we implemented the job queue using MySQL with the reasoning that a queue needs to be persistent even if the server crashes at one point during execution.

Queue Element

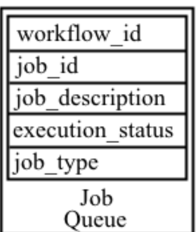

Figure 3.20: Content of a record

A record contains the associated workflow identifier, job identifier, job description, execution status, and job type (Javascript or WebCL). Workflow and job identifier will be sent along with the job description to the requesting remote worker so that upon receiving the result, the correct executor can be identified. The job description contains the code to be executed on the remote worker.

A remote worker will send a bit field to let the queue knows if it supports WebCL and WebCL jobs will only be distributed to worker capable of executing it.

Execution Status

The `execution_status` field belonging to a record can be one of the following:

1. `QUEUED`: indicates that it is in the queue and ready for execution.

2. `RUNNING`: indicates that it has been dequeued by a remote worker.

3. `EXECUTED`: indicates that it has been executed and result returned.

4. `ERROR`: indicates that the execution was not successful.

Accessing an Element
As the queue is shared by multiple workflows, care must be taken when accessing and modifying an element within it. Otherwise simultaneous access could cause race conditions and other unwanted consequences. In this case, we have protected such destructive access with table locking mechanism supported by MySQL.

3.8 Execution In Remote Workers

At the heart of WeevilScout lies the existence of remote workers. Whereas the server coordinates, remote workers are entities that perform the actual computations. A remote worker must be equipped with a browser that can execute Javascript. In addition, it must also support the WebWorker and (optionally) WebCL.

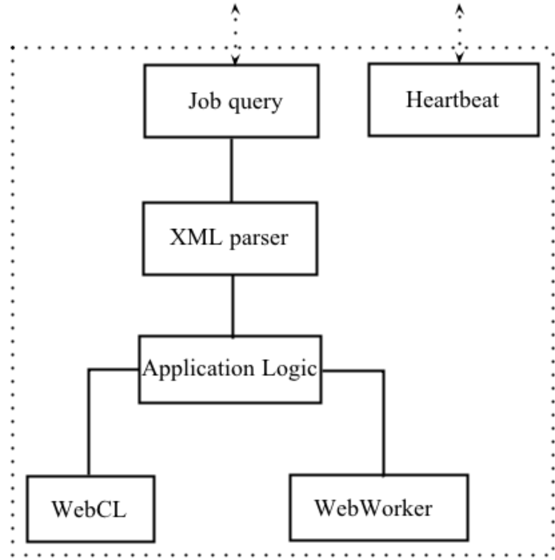

Figure 3.21: High-level architecture of a remote worker

Remote workers are assumed to be *unreliable* all the time. It can stop working at any time or calculates incorrect result either by deliberate act such as closing their browser or shutting down, or by unexpected events such as crash.

With this assumption we encourage *highly granular* workflows that consist of many small-sized jobs with the purpose of reducing the amount of work lost in the case where a worker stops responding.

Communication Protocols

Two protocols, the HTTP and WebSockets have been considered. HTTP is the de-facto standard but suffer from a relatively large overhead while WebSockets are very new, more efficient but needs additional component to be configured in both the browser and the server.

We have decided to only implement HTTP communication mostly due to its inherent simplicity and that our application does not do real-time communication.

Joining

A remote workers is said to be *connected* when it opens the URL a WeevilScout server. When it does so, it first supplies details such as browser type, WebCL capabilities, and location so that the server can recognize its capacities.

Dequeue

During the lifetime, a worker periodically asks for a job to be executed. This operation is called *dequeueing*. The server responds with a message that can be either empty if there are no jobs, or a triplet that contains the workflow ID, job ID, and the job itself.

A dequeue operation is performed periodically at either a fixed rate, or be made to depend on factors such as the worker's speed and the amount of work that has been done by that particular worker. In one case, dequeue frequency can be raised when the worker's machine is powerful enough to do more jobs. In another case, dequeue frequency can be lowered when a worker have done some amount of work as not to cause overloading, for example.

Submitting Result

Once a job has been executed by a remote worker, result will be sent back to the server.

Heartbeat

A worker must report back to the server periodically to indicate that it is still alive. When the server stops receiving heartbeat messages from a worker after a specified amount of time, the job that it currently process will be rescheduled.

Chapter 4

Evaluations

In this Chapter we describe the experiments that we did in order to evaluate the framework explained in Chapter 3. For this purpose, we have created two experiments, namely the Attained Performance Test (See Section 4.5.1) and the Pi-Approximation Test (See Section 4.5.1).

We ran the tests on two platforms, the SARA HPC Cloud and the Intel Core2 Duo PC. As a shared infrastructure, the attained performance on the SARA HPC Cloud suffers from fluctuations and might not reflect the maximum potential that can be achieved. Therefore for this reason, the Intel Core2 Duo PC is used to represent a stable platform with dedicated resources.

4.1 Metrics

FLOPS or Floating Point Operation Per Second was the main metric used in the experiment. As what constitutes one FLOP differs between one architecture to another, the results that we have gathered should be treated as a relative approximation rather than absolute figures.

Each experiment consisted of performing a fixed number of FLOP (Floating Point Operations) and the number of FLOPS can be estimated using the following formula,

$$FLOPS = FLOP\ count\ /\ elapsed\ time$$

Figure 4.1: FLOPS estimations

4.2 Profiling Framework

The profiling framework consists of two main components, one located in the remote workers, and another in the server. The component in the remote workers measures execution time compared to FLOP being performed by the use of a high-performance, accurate timer[1]. From these numbers we calculated the FLOPS from

[1]See `window.performance.now` (http://www.w3.org/TR/hr-time/)

the remote workers. These profile data are sent to the server and accumulated together with data from other remote workers. Each of these data is timestamped twice, once when a job is dequeued and once when it is submitted back to the server.

During the experiments, these collections of data with its timestamp was used for generating the visualization of a particular workflow execution.

4.3 Platforms

Below we describe the environment under which experiments were performed.

4.3.1 SARA HPC Cloud

The SARA HPC facility is provided for the Dutch academic community. At the time of writing it consisted of 19 nodes of Intel Xeon-E7 "Westmere-EX". Each node consisted of 32 cores processor running at 2.13GHz[48]. Access to these resources were provided via the use of virtual machines that runs on top of the KVM hypervisor.

The remote workers were configured in virtual machines, each with Ubuntu Linux (64 bit) and Mozilla Firefox 17.0. WebCL execution was not supported and therefore not tested on this platform.

4.3.2 Intel Core2 Duo PC

A personal computer was used to represent a somewhat more predictable environment. The machine consisted of an Intel Core2 Duo running at 2.26GHz. The machine supported WebCL execution through the use of AMD APP SDK and Nokia WebCL extension installed on Mozilla Firefox 17.0. There were two *OpenCL compute devices* that comes from the two CPU cores.

We performed the tests on two browsers, Mozilla Firefox 17.0 and Google Chrome 26.0 in order to compare the performances between different browsers. Mozilla Firefox 17.0 was used to perform both WebWorker and WebCL tests where Google Chrome 26.0 was used to perform WebWorker tests only, as it did not support WebCL.

4.4 Initial Benchmarks

We ran the following set of benchmarks on the Intel Core2 Duo PC platform in order to estimate performance of Javascript execution. The C programming language was used as the baseline. The benchmark set consists of four tests; The arithmetic, regular expression, and random number generation tests compare the speed of execution between Javascript and C, while the fourth test compares the speed of execution between WebCL and OpenCL.

Arithmetic Test

Test Type	Avg. FLOPS (JS)	Avg. FLOPS (C)	Factor
Arithmetic	3.5M	73.2M	0.04x

Table 4.1: Comparison of attained FLOPS of Javascript and C executions in arithmetic operations (larger is better)

The arithmetic test consisted of performing a large number of arithmetic operations over a certain period of time. The result indicated in Table 4.1 shows that Javascript execution is typically slower than C by a factor of 20.9, signifying that Javascript engines have yet to catch up with the speed of arithmetic operation execution achievable by native programs. Interestingly, the result from Section 4.5.4 indicates that we were able to attain much better performances when WebWorker was utilized.

Regular Expression Test

Search pattern	Searched text	Op per sec. (JS)	Op per sec. (C)	Factor
`xx*`	`(xo)*xx`	17302	1946	8.8x
`this|that`	`(thad)*this`	9127	2101	4.3x
`[1-5][1-5]`	`(17)*14`	9198	1766	5.2x
`a plain match`	`(a plain matc)* a plain match`	14212	1533	9.2x

Table 4.2: Comparison of attained FLOPS of Javascript and C regular expressions (larger is better)

On the contrary, Javascript performed much better than C when it comes to regular expressions. We performed four tests, each with different search terms as indicated in Table 4.2. Interestingly, Javascript performed faster than C in all tests by a factor of 4.3 up to 9.2, depending on the search terms used. For example, Javascript was only 4.3 times faster when the pattern contained alternations (Table 4.2, 2nd row). However, when performing a plain match (Table 4.2, 4th row) Javascript was 9.2x faster.

Random Number Generation Test

Test Type	Generation per sec. (JS)	Generation per sec. (C)	Factor
RNG	408,948	7,740,400	0.05

Table 4.3: The speed of random number generation in Javascript and C (larger is better)

The result in Figure 4.3 shows that Javascript was much slower than C when it comes to generating random numbers. In our test, Javascript could only generate roughly 400 thousand per second, compared to 7 million in C. In other words, Javascript was slower by a factor of 19 when it comes to generating random numbers.

WebCL versus OpenCL Performance Test

Test Type	Avg. FLOPS (WebCL)	Avg. FLOPS (OpenCL)	Factor
WebCL vs. OpenCL	65.1M	68.3M	0.95

Table 4.4: Performance comparison between OpenCL and WebCL (larger is better)

In order to compare the performance of WebCL and OpenCL, we created a test that consisted of performing a series of independent arithmetic operations in parallel. Execution of these operations were performed on the 2 instances of OpenCL compute devices on the Intel Core2 Duo PC platform. The result in Figure 4.4 indeed shows that the performance of OpenCL and WebCL was comparable, with WebCL only slightly slower than OpenCL by a slim margin.

This is to be expected as WebCL is only a wrapper that exposes OpenCL kernels to the browser environment[2].

4.5 WeevilScout Performance Evaluation

The following subsections describe the tests that we performed to measure WeevilScout capabilities, namely the Attained Performance Test and the Pi-Approximation Test. Each test consists of two versions, one used WebWorker jobs were used where the other used WebCL jobs instead.

4.5.1 Attained Performance Test

The Attained Performance Test is designed to measure the number of attainable Floating Point Operation per Second (FLOPS). The test is written in Javascript and WebCL. The goal of each test is to give impression of computing powers in the remote workers that can be harnessed by our system. Another goal is to show the difference in speed when WebCL is mixed in Javascript to allow hardware accelerated computation.

Pseudo-Code

```
1 function estimate_flops(n):
2     var sample_size = n
3     var flop_count = 0
4     var op = 0
5     for (i from 0 to n):
6         op = op + 13 - 10 * 14 * 2
7              * 15 * 2 * 4 * 2 * 3.6 * 1.2
8         flop_count += 10
```

Figure 4.2: Pseudo-code of the Attained Performance Test

The code in 4.2 consist of a loop that contains a series of arithmetic operations. The loop is executed n times, depending on the input.

[2]See Section 3.1

Workflow Coordination

The workflow structure consist of 1000 independent job node instances containing `estimate_flops` description. These jobs will be offloaded to remote workers and our monitoring framework will capture and aggregate the attained FLOPS on all participating workers.

4.5.2 Pi-Approximation Test

The distributed Pi-Approximation Test is used to show a somewhat more realistic application that does something meaningful and uses some features of the WeevilScout workflow engine.

It is an easily parallelizable algorithm that estimates the area under a circle. The algorithm works by generating dot at random point $(x,y) \parallel 0 \leq x \leq 1, 0 \leq y \leq 1$ repeatedly. This step is repeated many times until the number of point is large enough to produce an accurate estimation of the area under the circle. In our case, sample size as small as 10,000 gives an accuracy of two digits. However, for the sake of accurately measuring performance and minimizing the relative overhead, we opted for a much larger sample size of 1,000,000,000.

Pseudo-Code

The algorithm consists of two phases. The first phase is the generation of random numbers in parallel, where the second phase, which is the reduction phase, consists of gathering the results from the first phase.

```
1  function pi_approx(n):
2      var sample_size = n
3      var flop_count = 0
4      var count_point_inside = 0
5      for (i from 0 to n):
6          var x = random()
7          var y = random()
8          var z  = x*x+y*y
9          if (z <= 1) count_point_inside++
10         flop_count += 4
11     return count_point_inside
```

Figure 4.3: Pseudo-code of `pi_approx`

The algorithm consist of `pi_approx` function which generates a series of random numbers. This function will be instantiated in large number for a concurrent execution.

```
function pi_reduce(points_arr[], total_sample_size):
    result = 0
    for (elem in points_arr):
        result += i/total_sample_size*4
    return result
```

Figure 4.4: Pseudo-code of `pi_reduce`

`pi_reduce` works by aggregating all the points gathered from an array of `pi_approx` instances. The result returned from this algorithm will be the actual approximation of *pi*.

Workflow Coordination

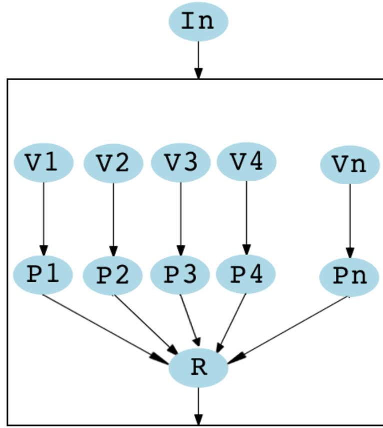

Figure 4.5: The workflow structure of Pi-Approximation Test

Figure 4.5 shows the workflow coordination employed for the Pi-Approximation Test. It is done using our *Pool collection node* which works as follows. First, an array of 1000 values will be used to start 1000 value nodes. Each of these value nodes contains the number of sample size that determines how many random points will be generated by the each job node. In total, there will be 1000 job nodes delivered to remote workers, each calculating random points for approximating the Pi value independently. After all jobs have finished executing, the result will be gathered and processed by a single reducer node instance.

4.5.3 Probing For The Optimal Sample Size

In WeevilScout, jobs are meant to run on a remote worker. However, both WebWorkers and WebCL imposes a significant overhead during start up. And

so, it is necessary for a job to execute long enough such that the overhead is comparatively low to the actual amount of work being done.

To determine this, we ran the Pi-Approximation Test several times with differing sample sizes, and compiled the result in Table 4.5.

Sample Size	MFLOPS (WebWorker)	MFLOPS(WebCL)
100000	2.66	1.20
500000	12.12	5.73
1000000	19.24	10.14
5000000	35.62	54.9
10000000	39.53	73.33
50000000	43.23	166.12
100000000	44.68	186.76
500000000	46.15	209.71
1000000000	46.49	211.36

Table 4.5: Probing for the optimal sample size where overhead is relatively minimal

We performed Pi-Approximation Test with sample size ranging from 100,000 to 1,000,000,000 with attained FLOPS which can be seen in Table 4.5. In the Pi-Approximation Test, sample size refers to the amount of iteration done which in turns affects the quality of the approximation. In other words the approximation should yield a more accurate result when the sample size is large.

Investigating the impact on performance when sample size changes was the main concern in this test; we were interested in determining how large should a sample size be so that the start-up overhead can be considered negligible. Note that the overhead from random number generations were not considered in this test. This is because the random number generators in WebCL have a very different characteristic and performances than the implementation available in WebWorker.

Tests were done using both WebWorker and WebCL jobs. Results indicates that when sample size is larger than 100 million, the attained FLOPS becomes large enough (44 MFLOPS for WebWorker jobs and 186 MFLOPS for WebCL jobs) that start-up overhead becomes relatively small. In effect, this translates to a job that runs approximately for 40 seconds in our Intel Core2 Duo machine.

4.5.4 Results From Intel Core2 Duo PC

This section comprises of results gathered from the Intel Core2 Duo machine which consists of running WebWorker and WebCL jobs.

Attained Performance Test (WebWorker)

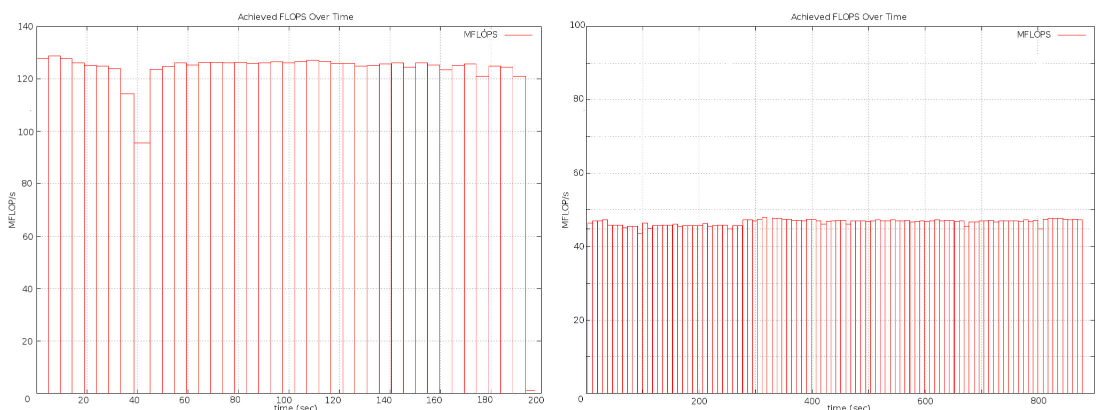

Figure 4.6: Google Chrome 24 (left) and Firefox 17 (right)

We ran Attained Performance Test with both Google Chrome and Mozilla Firefox to compare how these two browsers compare to each other in terms of performance, with the results shown in Figure 4.6. The results showed that Firefox was about 3 times slower than Chrome at 45 MFLOPS, compared to Chrome's 125 MFLOPS. It indicates that at the time of writing Chrome's implementation of WebWorker is more mature.

Pi-Approximation Test (WebWorker)

We ran another test to see how if the browser can exploit the available multi-core processors. The test that was executed on Firefox 17 compared the attainable FLOPS when opening WeevilScout in 1 tab compared to 2 tabs. By opening WeevilScout in two tabs, it was hoped that the worker can process two jobs simultaneously at a time, utilizing the two cores available in the Intel Core2 Duo PC.

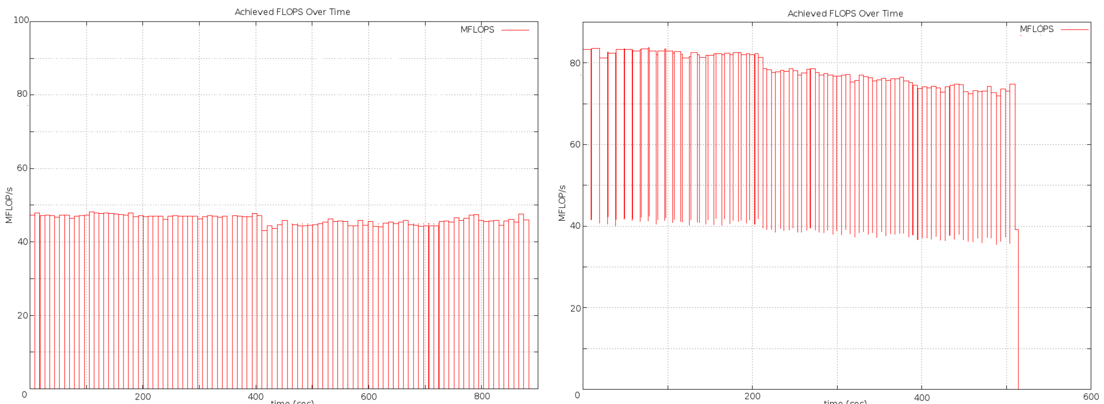

Figure 4.7: Execution in Firefox 17 with 1 tab (left) and 2 tabs (right)

Figure 4.7 shows that the results were positive - Firefox was again able to attain 45 MFLOPS when 1 tab was opened and 80 MFLOPS when two tabs were opened, indicating that Firefox were indeed able to exploit multi-core processors. There was a considerable overhead from these, as can be seen by the slightly diminishing FLOPS over time when opening WeevilScout in 2 tabs shown in Figure 4.7 (right).

Attained Performance Test (WebCL)

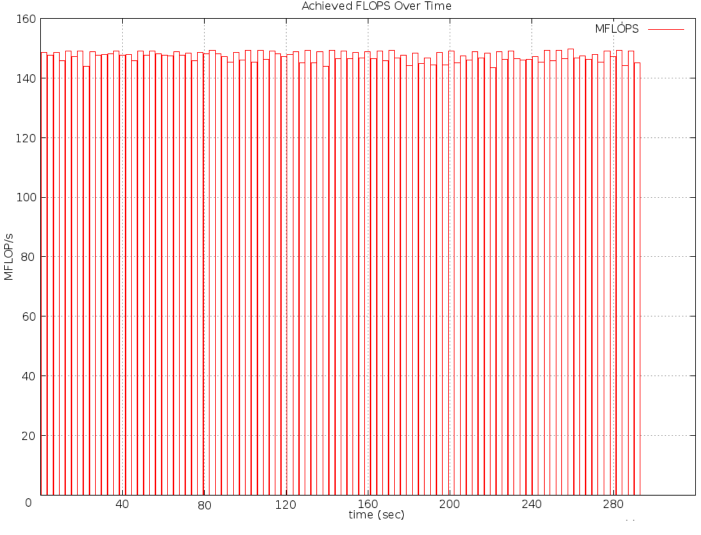

Figure 4.8: Achieved FLOPS of 1 worker in the Intel Core2 Duo PC platform

Figure 4.8 shows the result from Attained Performance Test in Firefox 17. We were able to gain a considerable amount of speedup, roughly 3 times faster at 145 MFLOPS, compared to the WebWorker job which averaged at 45 MFLOPS. It was partly due to OpenCL's ability to utilize every compute device available in the platform. In other words, unlike a WebWorker job which can only run on 1 core, a WebCL job can utilize all available cores.

Pi-Approximation Test (WebCL)

Again, this test was done in Firefox 17 as Nokia's WebCL plugin is not yet available on any other browsers. The test comprised of generating random points (with the use of a RNG) and a series of arithmetic operations. For this test, we used the MWC64X random number generator, which might perform much better when executed on a proper OpenCL compute device such as graphics card. In this platform we did not have a graphic card that is supported, relying on the 2 core processors as the only compute device.

Figure 4.9: Execution in Firefox 17 with 1 tab

The result depicted in Figure 4.9 shows that with WebCL, we were able to achieve roughly 75 MFLOPS in average. This is less than what we were able to achieve in the Attained Performance Test. Nevertheless, the result we gained yielded some insights about the performance of the execution of RNG in the WebCL platform.

4.5.5 Results From SARA HPC Cloud

This section comprises of results gathered from the Sara HPC Cloud. The test consist only of WebWorker jobs, as WebCL was not supported in the 64 bit OS that were present.

Attained Performance Test (WebWorker)

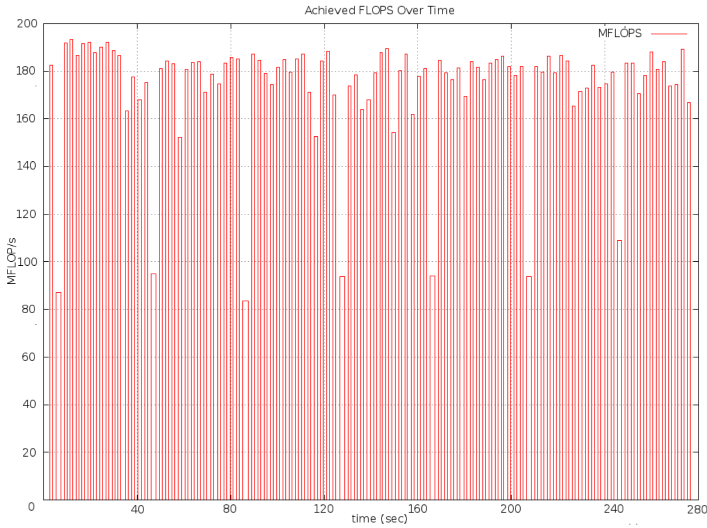

Figure 4.10: Attained Performance Test in SARA HPC Cloud with 1 worker

In Figure 4.10 we presented the attained FLOPS when only one worker was connected. This test serves as a guideline about the rough performance of each worker that was present in the HPC Cloud. A short job ensures that job execution finished faster than 5 seconds so that job execution would not overlap. We were able to achieve about 170 MFLOPS when all fluctuations, peaks and lows were considered. It is shown that attained FLOPS fluctuated between 170 and 190 MFLOPS.

However, it is also shown that the performance decreased dramatically for a very short amount of time every 40 seconds. This was most likely due to performing a refresh on the result page which consist of retrieving an XML string which contains details such as the status of each job in the job queue, link to job result should it be finished and time of execution. The XML would then have to be parsed and displayed as a table (using YUI's *DataTable* widget). We suspected that these processes caused a significant overhead that interfered with the performance of jobs.

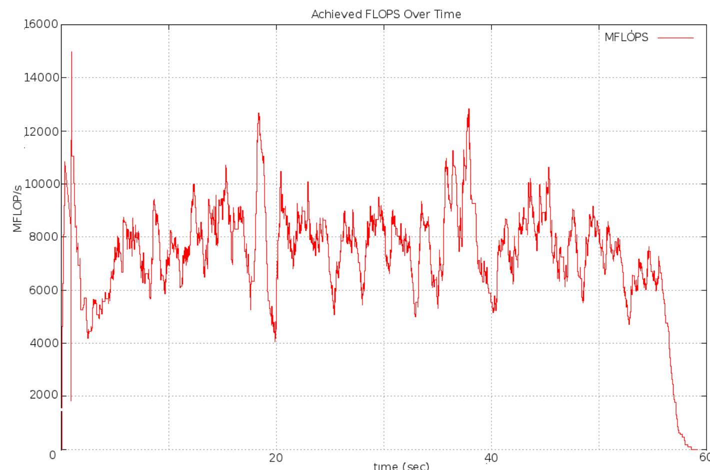

Figure 4.11: Attained Performance Test in SARA HPC Cloud with 72 workers

Result from heavy-usage scenario in which there were 72 workers is shown in Figure 4.11. At the start, there was a very high peak that reached up to 15 GFLOPS. The reason was that in the first 3-5 seconds most workers started and finished at almost the exact same time. After the peak, what followed was a rather significant dip as most of the workers that are finished by then were waiting for a new job to execute. Finally, the dip was followed by a relatively normal fluctuations which saw attained FLOPS averaged at around 7 GFLOPS.

Pi-Approximation Test (WebWorker)

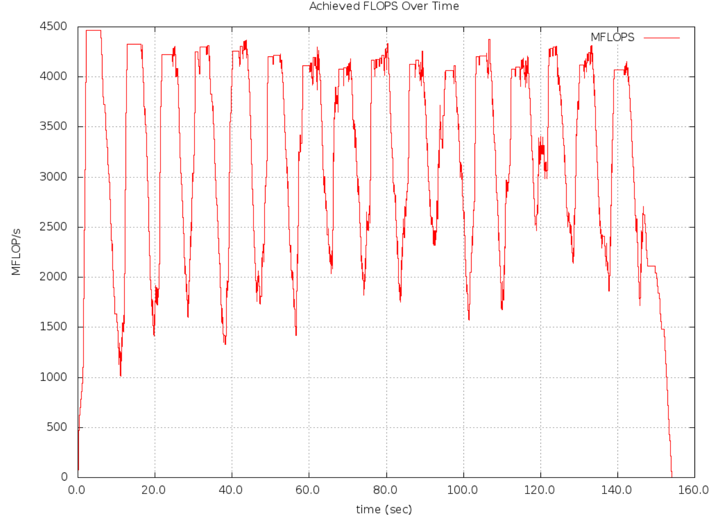

Figure 4.12: Execution of Pi-Approximation Test with 72 workers

Figure 4.12 shows the result of Pi-Approximation Test on the SARA HPC Cloud. Again, due to random number generation, attained FLOPS were not as high as in the Attained Performance Test. The dequeue time is set to *5 seconds*, as it normally was but in this test each job takes as long as *47 seconds* to finish on average. The worker would not dequeue a new job before the current one finishes as not to overload it. In this test we were able to attain around 4.5 GFLOPS. The pattern was much more recognizable, most probably due to the fact that jobs takes a while to finish.

The peaks and lows did not always happen at the time when a job was started or finished, so we would conclude that again updating the display of the result page seemed to take considerable amount of CPU resources. Another observation that we have made was that attained FLOPS per worker *decreases* when there are more connected workers. Our result showed that a worker should be able to attain roughly 130 MFLOPS. However, when we spawned 72 workers (each wrapped in its own virtual machine instances) the attained FLOPS in each of them dipped to about 80 MFLOPS. Seeing that there was no way for a worker to influence another (since there was no communication between them at all) it might be due to external factors such as how the hypervisor handles each virtual machine instances.

During this test, each browser averaged at about 80 MFLOPS. If each worker executes one job and another non-stop, theoretically the maximum attained FLOPS would roughly be 5.6 GFLOPS. With 5 seconds dequeue time, we were able to achieve 4.5 GLOPS in practice, meaning that we were able to utilize 80% of maximum attainable FLOPS for this particular scenario.

Chapter 5

Conclusions And Future Works

5.1 Conclusions

The performance of Javascript execution have definitely seen major improvement in recent years. Together with the emergence of hardware-accelerated computing enabled by WebCL, performing computation-intensive jobs on web-browsers have finally become a possibility.

However, simply distributing an arbitrarily large number of jobs to web-browsers would not produce any meaningful result. In this thesis project, we have developed a mechanism to coordinate the execution of jobs using the workflow approach. Our workflow implementation supports the dataflow models of computation where a node is activated upon receiving all of its required input. In practice, this is realized with our dataflow-based scheduler that schedules a job if and only if all of its required input has been met, allowing an output from one job to act as input to another.

Hardware-accelerated computation is supported with WebCL jobs. WebCL jobs enable true high-performance computation in WeevilScout, and are naturally suited for problems that are highly parallel by nature. As WebCL jobs can only be executed on OpenCL-capable devices, the WeevilScout server will determine if a worker supports WebCL and will only deliver them to workers that are capable of executing it.

Results that we have collected indicates that we were able to amass as large as 4.5 to 7 GFLOPS of computing powers when the tests that utilizes WebWorker jobs were executed on 72 workers in the SARA HPC Cloud. Although not performed on the said platform, theoretically even more powers could be harnessed should WebCL jobs were utilized, as the result from tests in the Intel Core2 Duo PC platform have indicated. Also, the result have indicated that our workflow implementation were able to coordinate the execution of a large number of jobs on an ensemble of remote workers successfully.

While browser-based computations have yet to achieve the efficiency of native programs, the advantage lies in the social aspect. Participating in a computation is only a matter of opening the project's URL, no matter what browser or operating systems they use. The fact that it is very easy for workers to participate makes it very easy to socialize projects built using WeevilScout, especially when combined with the power of social networking platforms that have seen widespread use in recent years.

5.2 Future Works

Obviously, as WeevilScout has yet to reach maturity there are many aspects open for improvements. Areas such as result checking, redundancy and fault-tolerance are crucial for ensuring the correctness of results. Communication mechanism will also have to be fine-tuned and made more efficient in order to accommodate for a large scale coordination of workers.

Increasing Communication Efficiency

In the current implementation of WeevilScout, workers communicate with WeevilScout server periodically, either to dequeue jobs or to inform the server that they are still responding. As the amount of messages that are communicated increases linearly with the number of connected workers, the performance of the server can be impacted negatively.

Our solution is to decrease the communication frequency between the workers and the server by an amount proportional to the number of connected workers. A constant *default* sets the default communication frequency, e.g. once every 10 seconds. The variable *threshold* governs how many workers are connected before the reduction can take effect. Another variable *rate* governs how much reduction should be applied.

The biggest impact of decreasing communication frequency is that workers would dequeue job less frequently, and in order to compensate, we propose a *batch dequeueing* mechanism. When reduced communication frequency is in effect, batch dequeueing mode will be activated, where workers will be allowed to dequeue and process many jobs at once.

It is important to consider whether to employ a *job overlapping* scheme where a job can be dequeued to more than one worker. When a worker communicates less frequently with the server, there is a bigger chance of it failing long before the server notices it. With this scheme, risk can be mitigated such that when a

worker failed to complete a job it is assigned to (due to crashing or other reasons), other workers that are executing the same job can serve as backups.

Fault Tolerance

Areas of improvement that needs to be considered includes the implementation of *fault detection* and *tolerance* mechanism. The WeevilScout server should employ techniques that detect incorrect results and deal with them, in some way. For example, most volunteer computing middlewares use *voting* based algorithm in which they send identical jobs to three or more workers, and if they return differing results the server choose to pick the *majority* as the winner.

Decentralized Communication

In the current design, the WeevilScout server acts as the centralized communication hub that coordinates the activities of all workers. As have been stated in the previous subsection, this approach bears the consequence of saturating communication channel to the server. Another approach that could be taken is the decentralized browser-to-browser communication in a *peer-to-peer* fashion. In this approach, a worker can contact the WeevilScout server to retrieve a list of existing *cluster* networks, of which it could opt to join. When a worker finishes computing a job, it could send the result directly to the worker within the cluster that requires the result.

Currently, peer-to-peer communication is not possible without depending on proprietary plugins such as *Adobe Cirrus*. However, with the advent of the WebRTC[56] standard it is hoped that peer-to-peer communication will be made possible in all browsers.

Parallel Job-Queues

The design of the job queue is limited in that it does not allow concurrent access. To circumvent this limitation, it is possible to create multiple queue instances where each instance can be accessed concurrently. This scheme will be most beneficial when the WeevilScout server is running on a multi-core machine. With this scheme, the optimal number of queue instances will depend on the number of hardware threads that are available on the said machine.

Chapter 6

Appendix

6.1 Job Examples

```
1 function weevil_main(a, b){
2     return a+b;
3 }
```

Figure 6.1: A WebWorker job that adds two values

```
1 function weevil_main(a){
2     a = eval(a)
3     var sum = 0
4     for (var i = 0; i < a.length; ++i) {
5         sum += a[i]
6     }
7     return sum;
8 }
```

Figure 6.2: A WebWorker job for summing all elements in an array

```
function weevil_main(a, b) {
    var platform_id = WebCL.getPlatformIDs();
    var context = WebCL.createContextFromType ([WebCL.CL_CONTEXT_PLATFORM,
        platform_id[0]], WebCL.CL_DEVICE_TYPE_CPU);
    var device_id = context.getContextInfo(WebCL.CL_CONTEXT_DEVICES);
    var command_queue = context.createCommandQueue(device_id[0],
        WebCL.CL_QUEUE_PROFILING_ENABLE);

    var intSize = 4;
    var elem = 1
    var a_tarray = Uint32Array(elem);
    var b_tarray = Uint32Array(elem);
    var ret_tarray = Uint32Array(elem);
    a_tarray[0] = a;
    b_tarray[0] = b;

    var a_mem_obj = context.createBuffer(WebCL.CL_MEM_READ_ONLY, intSize * elem);
    var b_mem_obj = context.createBuffer(WebCL.CL_MEM_READ_ONLY, intSize * elem);
    var ret_mem_obj = context.createBuffer(WebCL.CL_MEM_WRITE_ONLY, intSize * elem);

    command_queue.enqueueWriteBuffer(a_mem_obj, true, 0, intSize * elem, a_tarray, []);
    command_queue.enqueueWriteBuffer(b_mem_obj, true, 0, intSize * elem, b_tarray, []);
    command_queue.enqueueReadBuffer(ret_mem_obj, true, 0, intSize * elem, ret_tarray, []);

    clKernelStr = loadWebCLKernel("Add");
    var program = context.createProgramWithSource(clKernelStr);
    program.buildProgram([device_id[0]], "");
    var kernel = program.createKernel("Add");

    kernel.setKernelArg(0, a_mem_obj);
    kernel.setKernelArg(1, b_mem_obj);
    kernel.setKernelArg(2, ret_mem_obj);

    command_queue.enqueueNDRangeKernel(kernel, 1, [], [elem], [elem], []);
    command_queue.enqueueReadBuffer(ret_mem_obj, true, 0, intSize * elem, ret_tarray, []);

    command_queue.flush();
    command_queue.finish(); //Finish all the operations
    kernel.releaseCLResources();
```

6.2 Workflow Examples

```
1  <workflow>
2      <nodes>
3          <value id="val1" value="44" returntype="Number" />
4          <value id="val2" value="13" returntype="Number" />
5          <job id="val3" src="Add" returntype="Number">
6              <parameters>
7                  <parameter paramtype="Number" id="a" />
8                  <parameter paramtype="Number" id="b" />
9              </parameters>
10         </job>
11         <job id="val4" src="Add" returntype="Number">
12             <parameters>
13                 <parameter paramtype="Number" id="a" />
14                 <parameter paramtype="Number" id="b" />
15             </parameters>
16         </job>
17         <value id="val5" value="100" returntype="Number" />
18     </nodes>
19     <edges>
20         <edge id="first" source="val1" dest="val3" paramdest="a" /
               > <!-- 44 -->
21         <edge id="second" source="val2" dest="val3" paramdest="b"
               /> <!-- 13 -->
22         <edge id="third" source="val5" dest="val4" paramdest="a" /
               > <!-- 100 -->
23         <edge id="fourth" source="val3" dest="val4" paramdest="b"
               /> <!-- 57 -->
24     </edges>
25     <finalnode id="val4" />
26 </workflow>
```

Figure 6.4: A workflow description consisting of values and WebWorker jobs

```
<workflow>
    <nodes>
        <value id="val0" src="flop_test_sample_size" returntype="Array" />
        <pool id="pool0" returntype="Number">
            <parameters>
                <parameter paramtype="Array" id="a" />
            </parameters>
            <processor id="pool0_processor0" src="FLOPS" returntype="Number">
                <parameters>
                    <parameter paramtype="Number" id="a" />
                </parameters>

            </processor>
            <reducer id="pool0_reducer" src="FLOPS_SumReduce" returntype="Number">
                <parameters>
                    <parameter paramtype="Array" id="a" />
                </parameters>
            </reducer>
        </pool>
    </nodes>
    <edges>
        <edge id="edge0" source="val0" dest="pool0" />
    </edges>
    <finalnode id="pool0" />
</workflow>
```

Figure 6.5: A workflow description consisting of a value and a pool node

Bibliography

[1] T. Agerwala. Special feature: Putting petri nets to work. *Computer*, 12(12):85–94, 1979.

[2] G. Agha and C. Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In *Foundations of Software Technology and Theoretical Computer Science*, pages 19–41. Springer, 1985.

[3] M. Alt, A. Hoheisel, H.W. Pohl, and S. Gorlatch. A grid workflow language using high-level petri nets. *Parallel Processing and Applied Mathematics*, pages 715–722, 2006.

[4] D.P. Anderson. Boinc: a system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4 – 10, nov. 2004.

[5] D.P. Anderson. Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10. IEEE, 2004.

[6] Russell Atkinson and Carl Hewitt. Synchronization in actor systems. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 267–280, New York, NY, USA, 1977. ACM.

[7] Mark Baker. Cluster computing white paper. *CoRR*, cs.DC/0004014, 2000.

[8] A. Barth, J. Weinberger, and D. Song. Cross-origin javascript capability leaks: Detection, exploitation, and defense. In *Proceedings of the 18th conference on USENIX security symposium*, pages 187–198. USENIX Association, 2009.

[9] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and control*, 60(1):109–137, 1984.

[10] boincstats.com. Boincstats/bam! — boinc combined - detailed stats. `http://boincstats.com/en/stats/-1/project/detail`, 2013.

[11] Ulrik Brandes, Markus Eiglsperger, Juergen Lerner, and Christian Pich. Graph markup language (graphml). *Handbook of Graph Drawing and Visualization (Discrete Mathematics and Its Applications). Chapman & Hall/CRC*, 2010.

[12] T.F. Bresnahan and P.L. Yin. *Standard setting in markets: the browser war.* New York: Cambridge University Press, 2007.

[13] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration: A synergic approach for system design. *Service-Oriented Computing-ICSOC 2005*, pages 228–240, 2005.

[14] R. Cleaveland and S.A. Smolka. Process algebra. *Wiley Encyclopedia of Electrical and Electronics Engineering*, 1999.

[15] W.D. Clinger. Foundations of actor semantics. 1981.

[16] V. Curcin and M. Ghanem. Scientific workflow systems-can one size fit all? In *Biomedical Engineering Conference, 2008. CIBEC 2008. Cairo International*, pages 1–9. IEEE, 2008.

[17] R. Cypher and J.L.C. Sanz. *The SIMD model of parallel computation.* Springer-Verlag New York, Inc., 1994.

[18] G.A. Di Lucca, A.R. Fasolino, M. Mastoianni, and P. Tramontana. Identifying cross site scripting vulnerabilities in web applications. In *Web Site Evolution, 2004. WSE 2004. Proceedings. Sixth IEEE International Workshop on*, pages 71–80. IEEE, 2004.

[19] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S.L. Price. Grid service orchestration using the business process execution language (bpel). *Journal of Grid Computing*, 3(3):283–304, 2005.

[20] J. Ezpeleta, J.M. Colom, and J. Martinez. A petri net based deadlock prevention policy for flexible manufacturing systems. *Robotics and Automation, IEEE Transactions on*, 11(2):173–184, 1995.

[21] S. Frølund. *Coordinating distributed objects: an actor-based approach to synchronization.* MIT Press, 1996.

[22] A. Goderis, C. Brooks, I. Altintas, E. Lee, and C. Goble. Composing different models of computation in kepler and ptolemy ii. *Computational Science–ICCS 2007*, pages 182–190, 2007.

[23] A. Goderis, C. Brooks, I. Altintas, E.A. Lee, and C. Goble. Heterogeneous composition of models of computation. *Future Generation Computer Systems*, 25(5):552–560, 2009.

[24] Google. v8 - v8 javascript engine. `http://code.google.com/p/v8`, 2012.

[25] Ilya Grigorik. Collaborative / swarm computing notes. `http://www.igvita.com/2009/03/07/collaborative-swarm-computing-notes`, mar 2009.

[26] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. Evaluation of job-scheduling strategies for grid computing. *Grid ComputingGRID 2000*, pages 191–202, 2000.

[27] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[28] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[29] W. Jeon, T. Brutch, and S. Gibbs. Webcl for hardware-accelerated web applications. 2012.

[30] Saindane M. Kuppan, L. Ravan, javascript distributed computing system. `http://www.andlabs.org/tools/ravan/ravan.html`, 2009.

[31] McGrath R. Leonardo B., Huckestein J. Maprejuice, distributed computing via javascript. `http://maprejuice.com`, 2010.

[32] C. Leung and A. Salga. Enabling webgl. In *Proceedings of the 19th international conference on World wide web*, pages 1369–1370. ACM, 2010.

[33] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[34] H. Matsuno, A. Doi, M. Nagasaki, and S. Miyano. Hybrid petri net representation of gene regulatory network. In *Pacific Symposium on Biocomputing*, volume 5, page 87. World Scientific Press Singapore, 2000.

[35] R. Milner. *A calculus of communicating systems*. Springer-Verlag New York, Inc., 1982.

[36] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, and C. Goble. Taverna, reloaded. In *Scientific and Statistical Database Management*, pages 471–481. Springer, 2010.

[37] Mozilla. Spidermonkey — mdn. `https://developer.mozilla.org/en/docs/SpiderMonkey`, 2012.

[38] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46 – 52, oct. 2003.

[39] J.L. Peterson. Petri net theory and the modeling of systems. *PRENTICE-HALL, INC., ENGLEWOOD CLIFFS, NJ 07632, 1981, 290*, 1981.

[40] C. Reginald, G. Putra, A. Belloum, S. Koulouzis, M. Bubak, and C. de Laat. Distributed computing on an ensemble of browsers. 2013.

[41] C. Reis, A. Barth, and C. Pizano. Browser security: lessons from google chrome. *Queue*, 7(5):3, 2009.

[42] J. Ruderman. Same origin policy for javascript, 2009.

[43] Samsung. webcl - webcl for webkit. `http://code.google.com/p/webcl`, 2012.

[44] L.F.G. Sarmenta. *Volunteer computing*. PhD thesis, Massachusetts Institute of Technology, 2001.

[45] SVEN-BODO SCHOLZ. Single assignment c: efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13:1005–1059.

[46] A. Slominski. Adapting bpel to scientific workflows. *Workflows for e-Science*, pages 208–226, 2007.

[47] J.E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[48] SURFsara. System description of the hpc cloud. `https://www.surfsara.nl/systems/hpc-cloud/description`, 2012.

[49] Aarnio T. Webcl. http://webcl.nokiaresearch.com, 2012.

[50] I. Taylor, S. Majithia, M. Shields, I. Wang, and ID Config. D 3.3 triana workflow specification.

[51] W. van Der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques. *Business Process Management*, pages 19–128, 2000.

[52] W.M.P. Van Der Aalst and A.H.M. Ter Hofstede. Yawl: yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[53] R. van Glabbeek and F. Vaandrager. Petri net models for algebraic theories of concurrency. In *PARLE Parallel Architectures and Languages Europe*, pages 224–242. Springer, 1987.

[54] W3C. 7 w3c technical report development process. http://www.w3.org/2005/10/Process-20051014/tr, 2005.

[55] W3C. Web workers. http://www.w3.org/TR/2012/CR-workers-20120501, 2012.

[56] W3C. Webrtc 1.0 real-time communication between browsers. http://dev.w3.org/2011/webrtc/editor/webrtc-20120316.html\#peer-to-peer-data-api, mar 2012.

[57] S.A. White. Introduction to bpmn. *IBM Cooperation*, pages 2008–029, 2004.

[58] S.H.I.M.L.Y.G. Xin and X.Y.W.U.S. Guang. Wfms: Workflow management system. *Chinese Journal of Computers*, page 03, 1999.

[59] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *Sigmod Record*, 34(3):44, 2005.

[60] Jerry Zhao and Jelena Pjesivac-Grbovic. Mapreduce: The programming model and practice, 2009. Tutorial.