Universiteit van Amsterdam

Datafluo: A Scientific Workflow Engine

A thesis submitted to the Board of Examiners in partial fulfillment of the requirement of the degree of Master of Science in Grid Computing

Author:

Reginald Cushing

Supervisor:

Dr. Adam S.Z. Belloum

October 12, 2010

Contents

Abstract

A scientific workflow management system can be considered as a binding agent which brings together scientists and distributed resources. A workflow graph plays the central role in such a system as it is the component understood by both scientist and machine. Making sense out of a scientific workflow graphs is undoubtedly, the first and foremost responsibility of a workflow management system. Typical systems include an orchestration engine which models a workflow and schedules individual components onto distributed resources. As part of the WS-VLAM, we present an alternative orchestration engine which takes a different stand on interpreting the workflow graph. Whilst the current engine in WS-VLAM models the graph as a process network where components are tightly coupled through communication channels, the Datafluo engine models the graph as a dataflow network with farming capabilities. In this dissertation, we present the Datafluo architecture followed by a prototype implementation. The prototype is taken through its passes using scientific workflow applications where the generated results demonstrate the orchestration features. Through our results, we show how dataflow techniques reduce queue waiting times whilst farming techniques circumvent common workflow bottlenecks.

Chapter 1

Introduction

Over the past few decades the computer has proven to be an instrumental tool for scientists and their respective research, so much so that computational science branches have been introduced within different scientific domains. Through the new *in silico* paradigm of carrying out research, old frontiers were broken and new ones are being set. Breaking new frontiers is, amongst others, a matter of enhanced Science (e-Science) which can be defined as:

> *"e-Science is about global collaboration in key areas of science, and the next generation of infrastructure that will enable it."* - John Taylor [5]

Scientific Workflow Management Systems (SWMS) have become part of the science infrastructure in realising e-Science due to their intuitive approach in prototyping experiments while concealing underlying middleware complexities. SWMS are also instrumental in research collaboration since knowledge about experiments and data is easily shared through systems such as myExperiment [4]. This paradigm of designing, executing and sharing experiments enables scientists to focus on problem solving within their domain whilst intricate knowledge about underlying resources and workflow execution is hidden behind the SWMS. In essence, SWMS strive to bridge the knowledge gap between computational sciences and the myriad of distributed computing technologies. To date, many workflow systems have come to fruition and vary considerably in their workflow modelling, scheduling and targeted resources. The central component in a SWMS is the workflow. A workflow can be described as a connected graph which abstractly represents the flow of an experiment whereby the vertices represent the activities and the edges represent the dependencies between activities.

1.1 Motivation

WS-VLAM [40] is an e-Science Workflow Management System (SWMS) designed to orchestrate workflows using Grid environments. This system combines the ability to take advantage of the underlying Grid infrastructure and a flexible high-level rapid prototyping environment. From a high level perspective, a distributed application is composed of a data driven workflow where each component represents a process or a service on the Grid. WS-VLAM is being used for various scientific experiments including: *SigWin-detector [36]*, *Wave [10]*, and in *elaboration of molecular models [19]*. Thus, WS-VLAM is an asset which is continuously improved upon to meet the demands of the scientific community. Our practical experience in using WS-VLAM on Grid systems has led us to identify architectural areas that can benefit from enhancements to better accommodate multidisciplinary scientific workflows. Such areas include: the Model of Computation (MoC), the job submission management, and the component communication.

The current MoC models a stream-based process network represented by a directed acyclic graph (DAG). This type of model assumes that both parties of a streaming channel are able to synchronise on the channel thus communicating parties need to be active at the same time. In essence this couples the components in time which also has a domino effect on the rest of the workflow. For instance, if in a 3 component workflow (A, B, C) A is synchronised to B and the latter is also synchronised to C then A is inexplicitly synchronised to C as well. This model forces the internal system to schedule all workflow components at the same time. Such strong requirements encourage inefficient resource usage as tasks may lay idle while waiting for data. This is particularly evident with workflows composed of coarse grained tasks where a considerable amount of time is taken before data is produced on the output channels.

The current communication library is based on streaming peer-to-peer communication model. This further emphasizes the need for co-allocating all workflow tasks as no intermediary is available to buffer the communication. The current architecture is well suited for fine grained streaming workflows on single clusters where communication throughput is of utmost importance. In case of coarse grained scientific workflows running on the vast distributed resources such as Grids, guaranteeing co-allocation for large number of tasks is hard to achieve [20]. Also, since individual tasks are course grained, even if the co-allocation is assured, the time quantum allocated for tasks would probably elapse before leaf tasks start receiving data.

In this thesis, we describe the Datafluo engine for WS-VLAM. Datafluo is a dataflow approach with weaker scheduling constraints whereby each workflow task is scheduled dependant on data availability and hence alleviates co-allocation and encourages better resource usage. Since tasks are scheduled solely on the data availability, Datafluo does not impose any explicit task execution ordering and hence tasks are ordered at runtime according to the data. In our Datafluo approach, communication is decoupled in time through a messaging system. Tasks communicate through message exchange server hence alleviating the need for tasks to co-exist in order to communicate. Although such a messaging system will impose an extra communication overhead, we believe that the benefits such a system brings, namely, no need for co-allocation and ability for inter cluster task communication, are well worth the trade-off.

Chapter 2

Background

The e-Science term describes computational and data-intensive science. It has become a complementary experiment paradigm alongside the traditional *in vivo* and *in vitro* experiment paradigms. e-Science opens new doors for scientists and with it, it exposes a number of challenges such as how to organise huge datasets and coordinate distributed execution. For these challenges, a plethora of technologies and innovations have come together to enable e-Science. The following are some of the technologies and concepts aimed at e-Science.

2.1 Scientific Workflows

The knowledge gap between different scientific domains and computer science impedes the full exploitation of the distributed resources by scientific applications. For this reason high-level application tools are invaluable for scientists as they tend to abstract most underlying technological intricacies and let scientists focus on their problem domain. Scientific Workflow Management Systems are such tools. Workflows have been described as:

> *"The automation of the processes, which involves the orchestration of a set of grid services, agents and actors that must be combined together to solve a problem or to define a new service."* - [27]

In other words a scientific workflow is a formal description of a process for accomplishing a scientific objective which is expressed in terms of tasks and their dependencies [44]. Such scientific workflows can be part of any of the scientific life-cycle, be it hypothesis formulation, experiment design, execution and data analyses. Most often scientific workflow tasks are either computational or data intensive which makes them ideal candidates for distributed architectures as a single computer may prove to be infeasible to execute such workflows. The design and execution of a scientific workflow is the responsibility of a SWMS. The act of executing a workflow is commonly referred to as workflow orchestration and is

done by the SWMS engine, the main component in typical SWMS. Thus the main goal of a SWMS is to automate the workflow execution as well as provide support for design, monitoring, verification and collaboration.

2.1.1 Scientific Workflow Life Cycle

The scientific workflow life cycle describes the steps and phases which are supported by many SWMS. These steps are analogous to the traditional scientific methodology of carrying out experiments and can be categorised in *Design, Resource Planning, Execution, Analysis,* and *Dissemination.*

- **Design and Composition:** A scientist starts by defining the requirements for the experiment. Such requirements describe the major tasks that make up the experiment. Most SWMS include libraries of readily available workflows and tasks so the scientist would typically search a semantically annotated database for workflows and tasks to re-use within his experiment. Designing the new experiment involves either reusing or extending an existing workflow or creating a new workflow from scratch where tasks can be chosen from a library or developed specifically for the experiment. The resultant abstract workflow defines the processes involved and the dependencies between such processes. Common dependencies are data dependencies and control dependencies.

- **Resource Planning:** At this stage the scientist is allowed to perform additional steps on the workflow. These steps may include validation (e.g. type checking and loop detection in DAGs), graph optimisations (e.g. loop unrolling), resource binding, scheduling strategies (e.g. choosing between process network and dataflow models), data to be staged in and out, and parameter settings. After this stage the abstract workflow becomes a concrete, executable one which can be executed by a workflow engine.

- **Execution:** As for the SWMS, the execution stage forms the core part of the management system where the executable workflow is automated by the system. The main responsibilities for the execution engine are: staging in and staging out data, provide a communication substrate for task communication, execute tasks in accordance to the specified model of computation, monitor the execution, provide means of fault tolerance such as retry, and capture provenance information. Provenance information describes the workflow execution in a manner that the experiment can be repeated and

the same results obtained. Such information would include tasks executed, data consumed and produced, data dependencies, parameter settings and hardware information.

- **Analysis:** After the execution, the scientist would typically analyse the data so as to verify the results. Erroneous results could occur due to SWMS related execution errors, hardware related (e.g. floating point precision) and logical workflow errors. At this stage the scientist can opt to refine the experiment hypothesis and rerun the experiment to address the logical workflow errors.

- **Dissemination:** Dissemination involves sharing the results, workflows and metadata for further use and verification by the scientific community. Peer scientists should be able to rerun the experiment and verify the results. Efforts such as myExperiment [4] allow such collaboration.

2.1.2 Models of Computation

The *Model of Computation (MoC)* of a workflow system can be described as the logic behind interpreting the workflow graph and is a core part of SWMS execution engine. More formally, if we consider a directed workflow graph W which is composed of actors (vertices) and dependencies (edges), $\overline{p}$ the parameter set, $\overline{x}$ be the input data set, and $\overline{y}$ be the output data set, then the MoC M defines how to execute the workflow $W_{\overline{p}}$ on $\overline{x}$ input data to obtain $\overline{y}$ output data. Hence the MoC is the mapping $M : \mathcal{W} \times \overline{P} \times \overline{X} \rightarrow \overline{Y}$ for which a workflow $W \in \mathcal{W}$ with parameters $\overline{p} \in \overline{P}$ and data input $\overline{x} \in \overline{X}$ determines the output set $\overline{y} \in \overline{Y}$. Thus the output data set, $\overline{y}$, is defined as $\overline{y} = M(W_{\overline{p}}(\overline{x}))$ [44].

In many SWMS workflow graphs are *Directed Acyclic Graphs (DAG)* based. DAGs are graphs with unidirectional edges and no direct or indirect cycles. Such graphs are well suited to describe loosely coupled processes with data dependencies but are not expressive enough to describe control flow such as iterations where cycles are needed. For this reason some SWMS extend the DAG semantics to include explicit control structures which enable computation steering through conditional branching and iterations. Furthermore, the DAG model can be extended to include concurrency constructs such as parallel branches and task joins. The graph semantics coupled with its interpretation leads to the model of computation by a workflow system. The following are some common MoCs:

- **Process Networks:** In a Kahn process network[42, 39], sequential processes communicate through unbounded one-way *First-In-First-Out (FIFO)* channels. Channels carry an infinite stream of data objects or tokens. Tokens are only produced once and consumed only once. Writes to a FIFO channel are non-blocking while reads are blocked hence consumers will block until tokens are available for input. In a Kahn network, processes are said to be monotonic and continuous [42]. Monotonicity defines a process as being a streaming process where a process can act immediately on the input data without waiting for all the data before processing can start. Continuous defines a process that can continuously send data with respect to the input data. A Kahn process network is said to be deterministic although practical implementations tend to include non-determinism by allowing multiple processes to consume tokens from the same channel, multiple processes to produce tokens to the same channel, processes to share variables and implementing processes in a non-deterministic way.

- **Dataflow Networks:** Dataflow networks are a subset of Kahn process networks [42] which, in turn, are a computational model whereby concurrent processes communicate through unidirectional FIFO channels. In the Kahn model, writes are non-blocking, while reads on empty channels are blocking. In addition to this process network, a dataflow network adds the notion of *actors, tokens and firing rules*. An actor can be defined as a quantum of computation, a token represents a data parcel or message between actors and firing is the event that leads to the execution of an actor. FIFO channels are used to pass tokens between actors. Actor may fire only if tokens are present on every input port. Thus computation steering is dependent on the flow of data. Two common variants of the dataflow model are: synchronous dataflow (SDF), and dynamic dataflow (DDF). In SDF the actor firing rules are static and so allow fast implementations due to lack of decision making. On the other hand, in DDF, the firing rules are deduced at runtime. A major difference between the two is that in SDF, deadlock and communication buffer boundedness are decidable while in DDF they are not [30]. Classical dataflow modelling on grid infrastructures are particularly challenging since repeated task firing would result in repeated job submission which induces excessive overhead.

- **Discrete Events:** This model is a variant of a dataflow model. In discrete events, tokens are associated with a time stamp. Actors are not simply

executed when data is available but instead the coordinator sorts those actors according to the queued data age where actors having old data are given high priority. This will implement data fairness in a way that the oldest data gets to be processed first.

- **Synchronous/Reactive:** This model controls actors through a simulated global clock. On every clock tick all actors fire which, in turn will observe input values and assert output values. This model is analogous to processor architectures where functional units perform action on clock cycle edges.

- **Finite State Machines:** In an FSM, actors represent states. The coordinator stars with an initial state. States may have refinements in which case these are fired and evaluate guards on all output transitions. If any guard evaluates to true, the transition is taken and the state of the new actor becomes the new current state.

- **Demand Driven:** In this model execution starts from the leave nodes in the graph. The leave nodes will immediately block until all its parents are recursively initiated and start producing tokens. This model is well suited for large graphs with many end actors as only a sub-graph needs to be activated to produce results.

- **Petri-nets:** Petri-nets differ from DAG-based by modelling control flow and, most importantly, model state through the use of token transitions. Furthermore, Petri-nets' well understood properties such as deadlock and conflict can aid in model analysis and optimization.

2.1.3 Farming and Parameter Sweeps

Farming deals with splitting data n-wise amongst n identical tasks. This technique speeds up data processing especially when dealing with independent tasks. Farming can make better use of the resource by elastically replicating tasks to reduce empty resource slots while reducing the workflow makespan. If we consider data D to take time T_1 to process on one node and T_n on n nodes when dividing D amongst the n nodes, the ideal speedup is n and is defined as $\frac{T_1}{T_n}$. A close to ideal speedup can be achieved when assuming independent tasks with negligible communication overhead.

BOINC [38] is a task farming, CPU scavenging framework which has been popularised by SETI@home. BOINC is a centralised architecture where clients

log into servers asking for work. The BOINC system harnesses a wider distributed system through volunteer computing whereby any user on the Internet can take part in the system by donating computation and storage to be used by BOINC. Applications running on the BOINC system are largely independent and hence can scale quite well on such architectures.

Parameter sweeps are a special kind of task farming with the difference that the data being split amongst the task pool is the set of parameters. *Parameter Sweep Applications (PSA)* are characterised by an *embarrassingly parallel* application which is an application that can be decomposed into many independent tasks with little or no synchronisation or data dependencies. Parameter sweep model is a simple yet powerful concept used by many scientific application such as those in: computational fluid dynamics, bio-informatics, particle physics, discrete event simulation and computer graphics [14]. In a PSA, data is replicated to all collaborating tasks while each task is given a set of different parameters where each task in a PSA works on identical data. Since PSAs are intrinsically independent they can tolerate network latency and therefore scale to large distributed architecture. Additionally, they are amenable to simple fault tolerance mechanisms such as retries [14].

Nimord/G [11] is a system which aims at scheduling parameter sweep studies on grid architectures (Globus). Nimrod provides a declarative language for describing parametrised experiments. The core part of the architecture is a parameter engine which is responsible for parametrising the experiment, creating jobs and mapping tasks to the resources through a schedule advisor. The scheduling approach in Nimrod/G is based on grid economy with deadlines. This tries to achieve trade-offs between performance and cost [14]. Another similar application is AppLeS [14, 15], which focuses on the scheduling problem and provides various solutions such as *self-scheduled work-queue* and *adaptive scheduling with heuristics*.

Other parameter sweep efforts focus on extending known systems such as Kelper [44] to include parameter sweep capabilities. One such attempt proposes a master-slave architecture for the Kelper system [61]. The architecture targets networked computing resources. A master node initiates the workflow execution that manages a swarm of slave nodes to execute sub-workflows. Through this architecture slave nodes are able to run concurrently and process different data.

2.1.4 Task Scheduling

As with many SWMS that employ late resource binding techniques, the enactment engine is also responsibility for mapping the tasks to the suitable resources.

In such late binding systems, the MoC phase merely serves as ticking tasks as runnable. Mapping tasks to resources boils down to a scheduling problem. Finding an optimal solution for a schedule is a "NP-complete" problem and hence searching for the optimal solution for huge scheduling problems is infeasible. Most solutions incorporate other techniques such as heuristics, to find a near-optimal solutions. Some of the most common graph scheduling techniques developed over the years are: list scheduling [9], clustering [51], and task duplication [9].

List scheduling is a two step process: first, each node in the graph is assigned a priority which could be derived from heuristics. The nodes are ordered according to their priority. The node with the highest priority is made runnable. The second step involves choosing the best possible resource which allows the earliest start time [9]. Task duplication aims at better resource utilisation and involves transforming graph forks into parallel sub-graphs where the parent is duplicated for each child. This schedule has the effect of reducing waiting time by filling resource time slots with duplicates which also reduce the total execution time [9].

Clustering is yet another directed acyclic graph scheduling technique. It is known to be a "NP-hard" problem [51]. Clustering graph techniques are mostly based on weighted directed acyclic graphs for which each node and edge carries a corresponding weight. Node weights represent an execution metric while edge weight represents the communication metric. The clustering technique aims at optimising locality since it assumes that closer tasks incur less communication overhead. Close proximity tasks would generally mean that tasks execute on the same local network or local machine hence can exploit fast messaging fabric. Clustering can also include other techniques such as duplication which produce schedules with shorter makespan.

If we consider scheduling as a multiple space search problem, then most algorithms try to optimise one space such as time, cost or load. Most algorithms focus on time optimisation where the schedule tries to minimise the total workflow execution time or makespan. Such algorithms include the *Heterogeneous Earliest Finish Time (HEFT)*[64] which orders workflow tasks by ranking them against the predicted execution time and data transfer time. The ordered list is then mapped to the appropriate resources. Other simple schedulers are *Least Used Resource* which maps successive tasks to the least used resource, *Round Robin* which optimises the resource load by distributing the tasks evenly on all resources, and myopic *Just In Time (JIT)* schedulers which simply schedule tasks with little or no optimisation. Complex scheduling algorithms such as genetic algorithms tend to optimise multi-dimension search spaces by applying the principle of evolution[67].

Genetic algorithms maintain a population of solutions. Each solution is referred to as a chromosome. On every iteration the solutions are filtered through a fitness function and the best are combined together to form the new, better population. The iterations continue until the fitness function selects a suitable solution or an iteration counter threshold has been met.

A distinction is made between static and dynamic scheduling within the context of distributed architectures [28]. In dynamic scheduling tasks are scheduled at a late stage and decisions are usually taken on a per task bases. This strategy is most effective with dynamic workloads and volatile environments where the future workload is unknown and the environment is characterised by a high churn rate. On the other hand, when resources are stable, static scheduling can produce optimised efficient schedules since little decisions are taken at runtime.

2.2 Computing Paradigms

The exponential Internet growth rate coupled with advances in network technologies have made resources (e.g. computation, storage and special hardware) accessible around the world. Efforts to harness such distributed resources have lead to different yet overlapping paradigms.

- **Grid Computing:** grid computing paradigm [23] deals with globally dispersed, heterogeneous, loosely coupled resources. Resources in a grid architecture belong to different administrative domains and are managed separately. Resources and users are grouped together into *Virtual Organisations (VO)* where institutions within a VO share resources amongst each other. Resources in a grid architecture are made available through middlewares that specifically expose the resources to be shared while also handling communication, data management and security context. Middlewares such as Globus and gLite are a set of tools that manage job submission, monitoring, fault tolerance, data movement, and security. These tools implement the *Infrastructure as a Service (IaaS)* model where computing hardware is directly shared. Service oriented architectures within the grid such as *Web Services Resource Framework (WSRF)* services implement a *Software as a Service (SaaS)* model where software services are exposed. *Enabling Grids for E-sciencE (EGEE)* is one of the largest grid architectures based on gLite middleware. To date, EGEE [1] resource pool include 92,000 CPU cores and several Petabytes of storage. The resources are grouped in over 200 different virtual organisations and are used on a daily basis by thousands of scientists.

- **Volunteer Computing:** Volunteer can be considered as the desktop lightweight grid. In volunteer computing users decide to donate their idle processing power (CPU, GPU) and excess storage to distributed projects such as SETI@home and Folding@home. The most widely used middleware for volunteer computing is the BIONIC system which is a centralised system for CPU scavenging. Client report in for work at a central server which in turn distributes the workload amongst the clients. Applications for such environments are usually characterised as being farmed or parameter sweep types.

- **Utility Computing:** With utility computing organisations do not manage their in-house clusters but instead subscribe to a service provider and pay only for the hardware and software they use. Service providers are able to sell some of their computing power since most resources will be over-provisioned to deal with peak demands. Utility computing targets the *Platform as a Service (PaaS)* [23].

- **Cloud Computing:** Cloud computing paradigm overlaps utility computing but in cloud computing the focus is on the *Software as a Service (SaaS)* [23]. This allows programs to exist in the cloud as opposed to one's personal computer. The advantages for such a paradigm include: service outsourcing such as mail service, fault tolerance, and better security.

- **Edge Computing:** In edge computing [43], the aim is to push core services out to the network fringe. Having content and services accessible closer to the user will reduce network load and latency. The challenge in many edge technologies is the data coherency model where data replication techniques across distributed caches play a vital role to keep the data view consistent.

2.3 Coordination Patterns

An important aspect of any distributed system is the communication model it supports since distributed tasks will eventually rely on communication to synchronise and coordinate execution. Inter process communication on single machines is made simple through signals and shared memory where latency and bandwidth are relatively negligible. On the other hand communication between distributed process is a breed apart since communication relies on networking technologies

which is characterised by relatively high latencies and low bandwidth. Furthermore, distributed processes, often, act on private local data and hence challenges arise to maintain a data consistency model between distributed processes.

Distributed process communication models may vary considerably. Common models such as *remote procedure call (RPC)* aim at abstracting the low level communication layer by implementing a RPC API where remote procedures are called as if they are local to the system. Another common model is *message passing interface (MPI)* which encapsulates communication into messages and disseminates them amongst processes. At a lower level, communication may employ additional systems to enforce *quality of service (QoS)* since common protocols such as TCP and UDP do not guarantee.

Communication in distributed systems goes beyond data exchange as it also plays a role in coordinating the execution of the whole system. Coordination models can be broadly divided into four categories [56] as shown in figure 2.1. The models are organised according to their temporal and referential coupling. In temporal coupling both communicating parties need to be active during communication while referential coupling means that communicating parties know beforehand with whom they are communicating. The strictest category is a coupling in time and reference as would be with direct TCP/IP communication where both parties need to be active during communication and are addressable through IP addresses. Transient message passing and direct peer-2-peer systems fall within this category.

Decoupling communication in time would have the effect of allowing processes to communicate at any time independent of the execution time. This characteristic is analogous to a mailbox system where processes leave messages in a mailbox which are later picked up by other processes. Mail messages are addressable hence this communication model is coupled in reference since communicating processes need to address mail messages to a particular process or a set of processes.

Decoupling communication referentially means that processes do not know beforehand who will read the messages. Communicating parties are coupled in time so processes need to be active and running during communication. The model is analogous to an event based system where processes fire events which in turn interrupt other processes. The listening processes need to be running at the time of the events hence the time coupling.

Decoupling communication referentially as well as in time results in generative communication which does not depend on processes being active at a certain time and communicating parties also need not know explicitly to whom they are

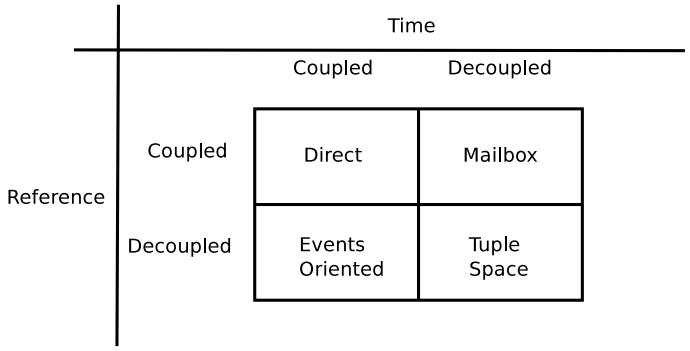


Figure 2.1: A taxonomy of coordination models [56]

communicating. These types of communications are exemplified through tuple space communication such as publish/subscribe systems where processes write data to a tuple irrespective of who will read it and when it will be read. Tuple spaces where popularised with the Linda model [62]. Linda implements tuple spaces as a global associative memory with basic operations such as: `in` which reads and removes tuples, `rd` which only reads tuples, and `out` which writes new tuples.

2.4 Grid Task Management

Grid architectures are commonly described as an abstract layered architecture [26] as depicted in figure 2.2. The layered architecture is somewhat comparable to the Internet protocol stack where the top three layers map to the Internet application layer while the connectivity and fabric layers map to the transport, Internet, and link layers. Task scheduling and management in grid architectures is usually a two step procedure and takes place at the collective layer and resource layer.

The collective layer task scheduling is a super scheduler and deals with multiple grid site scheduling. Such a scheduler is the gLite's *Workload Management Service (WMS)* [21]. The main components in the WMS are: the task queue, the information supermarket which is a database of resource information such as load, architecture and installed libraries, the matchmaker which uses the information to match tasks to resources, and the actual job submission and monitoring component. The matchmaking component takes decisions depending on many factors including: resource availability, specific user requirements such as the number of processors and software libraries needed, and grid site utilisation policies imposed by the local site administrators. Furthermore, the WMS can adopt different matchmaking strategies such as eager scheduling where the task is matched to a

resource as soon as possible and sent for execution immediately, and lazy scheduling where the scheduler waits for the resources to become available which are then matched against the list of tasks. The best matched task is then submitted to the resource.

Whilst the collective layer scheduler has a bird's eye view of the whole grid resources, the local resource layer scheduler is specifically aimed at optimising the schedule on the local cluster. Schedulers such as Maui [3] include extensive scheduling features such as multi task prioritisation, advance reservation, task *Quality of Service (QoS)*, and backfill scheduling techniques which can start lower priority tasks in an *Out-of-Order (OoO)* fashion without interfering with the high priority tasks. Another popular cluster workload manager is the *Sun Grid Engine (SGE)* which implements similar functionality to Maui.

As the critical path of a job submission takes it through two different schedulers, it is to no surprise that queue waiting times play a role in overall grid job execution time and performance. For this reason a number of tools aim to alleviate the queue waiting time through the notion of pilot job submission. The idea of pilot job submission is simple yet effective. Pilot jobs are special jobs that, when submitted and eventually run on the resource, they will pull the actual job for execution. This paradigm offers some advantages: by priming the grid with such pilot jobs, one can hide the queue waiting time since a number of pilot jobs would be available to immediately pull and execute a task. The pilot job can also perform some initialisation routines such as check the node environment and can accept jobs only if the node is validated hence increasing the job success rate. A networked collection of smart pilot jobs can build a virtual grid dedicated to one user and can provide additional features such as job monitoring. Common pilot-based systems are DIRAC [59] and Glide-WMS [54].

2.5 State of The Art in SWMS

In order to benefit from previous experiences in the field of scientific workflow management systems, we conduct a review of existing popular SWMS to gain insight on workflow enactment engines. Several well-known systems have been reviewed include: ASKALON [24][53][18], GridNexus [37], GWES [34], Karajan [60][18], Kepler [44][22] , Pegasus [13][18], Taverna [50][18] and Triana [57][58][31].

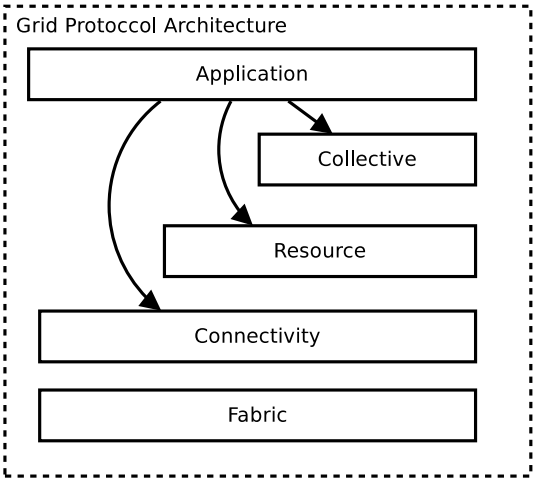


Figure 2.2: Layered grid Architecture analogous to the Internet protocol stack[26]. The fabric layer deals with the actual resources such as computation and storage. The connectivity layer defines communication and authentication protocols for grid specific network transactions. The resource layer defines protocols for resource management on single resource. The collective layer deals with multiple resources. The application layer describes user application grid interfacing such as APIs.

2.5.1 Kepler

Kepler [44] provides a graphical workflow composer whereby workflow components are *actors*. The scientist builds workflows by either reusing components or by composing the workflow from scratch. The designed workflow can be directly executed from within the GUI or else exported to an XML format(MoML) which could be passed to the Kepler execution system without the presence of the GUI. Kepler system is actively adapted and extended by the community. Kepler was extended to support dynamic component embedding by introducing a *frame* actor which is a dumb actor that gets embedded at runtime [47]. During execution stage, Kepler actors can be tagged as compute intensive. These actors are distributed across remote computation resources which include Kepler peers or Kepler slaves. Kepler execution engines determines the dataflow schedule and job execution based on the abstract workflow schedule. During design stage the scientist can indicate different *Models of Computation (MoC)* through different *directors*. Such common MoC are: Process Network, Dataflow, Discrete Events, and Synchronous/Reactive. Kepler includes a provenance recorder which can record data for all actors or just a subset. A provenance query API is then used to query the current workflow provenance store hence enabling monitoring whilst also allowing authorised users to query past provenance data.

2.5.2 Taverna

In Taverna [50], users compose hierarchical workflows in a graphical composer. The workflow nodes represent web services while edges represent the dataflow between the web services. The graphical representation is synthesised to an XML-based DAG format language, SCUFL. Execution in Taverna depends on the user mapping each workflow graph node to an available web service or set of web services. The latter allows for an *alternate* scenario when dealing with fault tolerance. Fault tolerance is also handled with retries whereby the task is retied a number of times upon failure.

2.5.3 Triana

As in other systems, Triana provides a graphical workflow composer. Triana interfaces to a different execution environments through GAT and GAP. The former is used to interface task oriented workflows to grid middleware such a Globus by using GRAM, GRMS or Condor. GAP is used for web service oriented workflows by binding to WSRF, Web and P2P services.

2.5.4 Pegasus

Pegasus [13] has no real graphical user interface. Pegasus can be considered as an execution engine which excepts XML (DAX) workflow specifications and executes them. DAX can be composed by separate user interfaces such as Wings. Pegasus workflow management system is a three part system composed of the Pegasus mapper, DAGMan execution engine and the Condor task manager. The mapper maps the workflow on resources managed by different managers such as PBS, LSF, Condor and individual machines. The mapper produces and executes workflows with directives to DAGMan on how to execute the workflow components. Directives include remote job execution, data movement and data registration. DAGMan is responsible for scheduling the components for execution. Depending on the input task graph, it submits jobs to Condor or Condor-G. Condor-G enables Pegasus to interface to Globus. DAGMan is also responsible for task-level fault tolerance whereby it retries failed jobs and creates rescue DAGs. The rescue DAG contains the portion of the original workflow which had not been executed. Pegasus captures provenance during execution which includes information such as host, runtime and environment variables.

2.5.5 Karajan

Karajan [60] is a Java based workflow management system which evolved through GridAnt. The basic system components are a visualisation, check-pointing and execution subsystems. Workflows are described in an XML-based language. Components are composed in a hierarchical fashion using a DAG model with extended primitives. Primitives provide generic sequential and parallel execution, sequential and parallel iterations (non-DAG), conditional execution and functional abstraction. Karajan interfaces to different grid middlewares through an abstraction API. It also supports late binding hence deferring the decision of how a task should be executed until the task is actually mapped to a resource. The system provides user directed and global fault tolerance. A noteworthy feature in this system is the fact that Karajan can be extended through parametrised user-defined workflow elements or by implementing new workflow elements.

2.5.6 GWES

GWES models grid workflows using Petri-net-based graphical modelling. Petri-net modelling differ from the commonly used DAG modelling in control flow and state. Petri-nets can model loops while pure DAGs do not. Control flow modelling in DAG is usually achieved by extending the model. Also, Petri-nets model graph/net state through token transitions on the other hand DAGs only describe the behaviour of the graph and do not capture state. Petri-net's well understood properties such as deadlock and conflict can aid in model analyses and optimisation.

2.5.7 ASKALON

ASKALON uses an XML based language *Abstract Grid Workflow Language (AGWL)* to describe the workflow graph. AGWL models DAGs with control structures such as sequences, loops, conditionals and advanced constructs such as parallel and collection constructs. The language also supports modularisation and reuse by defining sub-workflows. A core component of this workflow system is the scheduler. The scheduler consists of three sub-components; workflow converter, scheduling engine and event generator. The workflow converter is responsible for converting the abstract workflow into a simple DAG workflow by unrolling loops. The simple DAG can then be scheduled using graph scheduling algorithms. The scheduling engine maps the actual simple DAG to the underlying resources. The

scheduler has three different algorithms namely; *Heterogeneous Earliest Finish Time (HEFT)*, a genetic algorithm and myopic *Just In Time (JIT)* algorithm. The events generator is responsible for monitoring the workflow execution. Fault tolerance is achieved at three different levels; activity-level by means of retry and replication, control flow level through check-pointing and migration, workflow-level through alternate task, redundancy and check-pointing.

2.6 Current WS-VLAM Architecture

The current WS-VLAM architecture is composed mainly of two parts, the WS-VLAM composer implemented as a client application and a set of WSRF services deployed in GT4 Container. The service part of WS-VLAM architecture consists of a number of WSRF services. Some of them are standard services provided by GT4 Toolkit such as the Delegation Service, and a set of WSRF services developed in the VL-e project namely: the Workflow Components Repository (WCR), the Resource Manager (RM), the Runtime System Manager (RTSM).

Modules are the core composition entities for WS-VLAM. They represent tasks to be executed on the grid and are represented as vertices in the workflow graph. Tasks can be applications specifically written for WS-VALM, web services or legacy applications. Modules have input and output ports which interconnect the whole graph. Web services and legacy applications are supported through a system of wrappers which are specialised components aimed at integrating third party applications.

The execution engine is wrapped as a Run Time System Manager (RTSM) Service. This engine consists of two parts: RTSM Factory Service and RTSM Instance Service. The factory is a persistent service which instantiates a transient RTSM Instance service whenever a user submits a workflow execution. The RTSM is a super component composed of three sub-components: module launcher, module connector, and module controller. The module launcher initiates the workflow execution after parsing the WS-VLAM DAG XML description. Modules are submitted through a GT3 launcher. The module connector is responsible for setting up the data channels between the submitted modules. For each data channel the connector selects a TCP port and sends commands to the communicating parties which in turn setup the connection. The communication protocol is set to the GridFTP protocol which is a reliable transfer protocol suitable for large data transfers. The module controller monitors the executing module and reports back any events as well as state changes. WS-VLAM models a process network model

of computation where modules are initiated simultaneously and block on reading empty channels.

Module communication is maintained by the `libvlport` library which enables stream data communication between different modules on different nodes. Implementation-wise, modules implement a `vlmain` interface function which represents the computational logic. The `vlmain` function gets called by the API to initiate the module. The life-cycle of a typical module is as follows:-

- **Module Initialisation:** RTSM submits a module to grid-enabled resource using GRAM protocol or to the local node. Initialisation takes care of setting up the environment.
- **Port Creation:** Module input and output ports are created to allow module communication.
- **Registration:** Upon startup modules register themselves to the RTSM. Keep-alive messages are sent to the module during runtime to assert the modules lifelines. If no acknowledgements are sent back, the module is deemed dead.
- **Module connection:** The RTSM connects module output ports to input ports which in essence build the workflow graph at runtime.
- **Scheduling:** RTSM schedules the module for execution by calling the `vlmain` function on the module. This implements the actual module scientific logic.
- **Execution:** Modules read data from input ports, process the data and output onto the output ports. Typically modules will loop on input ports until the data stream is exhausted.
- **Termination:** Upon termination, the module exits the `vlmain` function, ports are closed, buffers are flushed and the module unregistered itself from the RTSM registry.

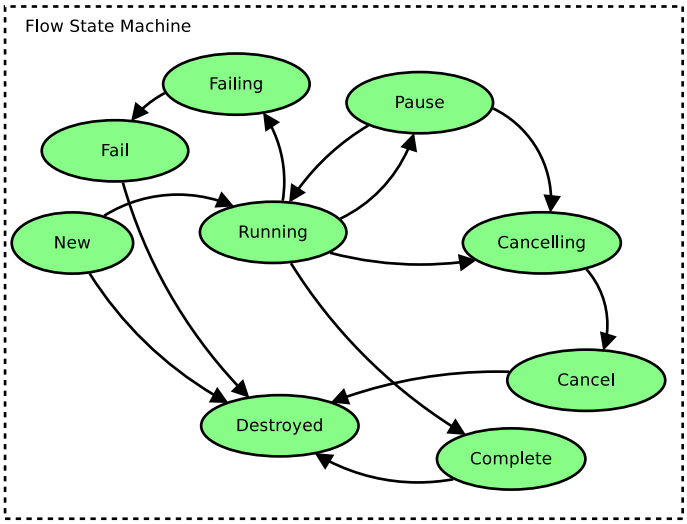
WS-VLAM supports parameter sweeps at a workflow level. A range of parameters can be set which would in turn initiate multiple workflow instances for each parameter. As a matter of scalability the RTSM engine supports distributed hierarchical execution engines. Workflow modules can be composed of either atomic modules or composite. A composite module describes a sub workflow which is

managed and executed by a separate RTSM instance. This allows instances to coordinate sub workflows as opposed to one engine responsible for the whole workflow execution.

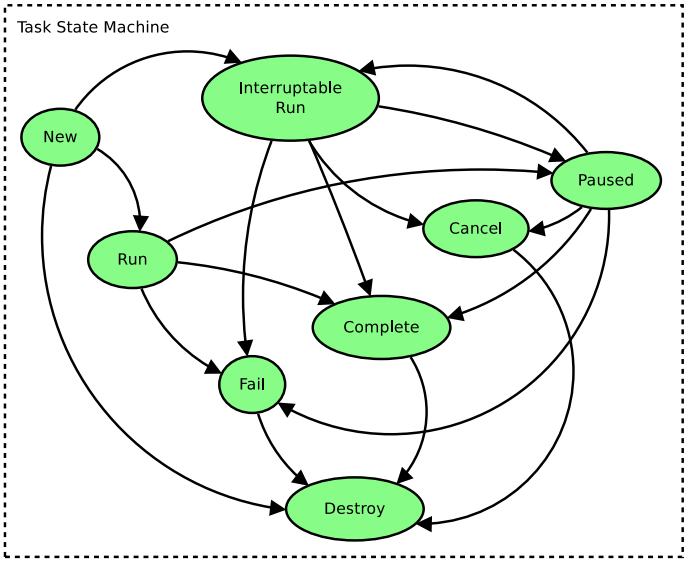
2.7 Freefluo

Freefluo [2] is a flexible Java workflow orchestration tool aimed at web service workflows. It was part of the Taverna [35] workflow system's orchestration engine. Freefluo is not tied to any specific workflow language or execution architecture, instead it provides a set of tools that enable extensions for specific workflow languages and execution models. Freefluo defines two major objects: a flow, and a task. A flow represents the workflow graph and is a collection of tasks (vertices) and connections (edges). Tasks are organised into three sets: the start tasks set, the end tasks set, and the set of all tasks.

The flow and task objects have associated state machines. The state transitions are illustrated in figures 2.3(a) and 2.3(b). Freefluo models state transitions as events. Each flow and task object have an associated state object which track the objects state changes. Objects interested in state changes implement a state listener which is a callback function called by the flow or task on every state changed. Thus a flow engine listens for state changes on each task to coordinate the flow execution. The Freefluo enactment engine models a DAG workflow. Upon startup, the engine invokes simultaneously the tasks in the start set. The start set is the set of tasks with no inputs. Running a task involves: changing the task state, and calling the `run` method on the task object. Upon completion, the task updates its state to a complete one which in turn triggers the engine to run the respective dependent tasks. This procedure continues until all tasks in the end task set complete and hence the whole workflow completes. The core of the enactment engine contained inside a workflow instance is decoupled from the actual invocation mechanism. This allows the engine to orchestrate a workflow in a generic way, while making it possible for tasks to invoke a variety of resources (from local processes to grid jobs).



(a) Freefluo flow model state machine



(b) Freefluo task model state machine

Figure 2.3: State machines for Freefluo major object models: flow, and task.

Chapter 3

Datafluo Architecture

This chapter describes the proposed Datafluo architecture. The architecture is a new engine for WS-VLAM and intended to work side by side the current RTSM engine. Datafluo name is a corruption of the term dataflow and the Freefluo [2] workflow engine on which Datafluo is based. Figure 3.1 illustrates a high-level overview of the Datafluo server and client side architectures.

3.1 Enactment Engine

The Datafluo enactment engine is the entry point and pivotal component in the whole system. It accepts a WS-VLAM DAG XML workflow representation, generated by the graphical workflow composer. The WS-VLAM XML workflow describes the workflow graph in full and contains information such as: task names (vertices), connections (edges), task parameters, and user defined connection types. At this stage the DAG is interpreted and a Datafluo object representation is generated. This representation dictates the MoC for the entire system. We define our MoC as a pipelined dynamic dataflow process network. As described in section 2.1.2, dataflow networks are based around the notion of *actors, tokens and firing rules*. In our architecture, actors are analogous to tasks, components or modules, tokens represent communication in the form of messages, and firing-rules are the events within the enactment engine that orchestration the workflow. Modelling a pure dataflow model on a distributed architecture would mean that actors are destroyed and re-scheduled on every respective firing event. This will undoubtedly incur excessive overhead due to the time taken to re-schedule a task on the resource. For this reason, we included pipelining in the model and deviated from the pure dataflow model so as to mitigate this problem.

In our Datafluo model we combine the concept of pipelining to the dataflow model by setting up a pipelined process network using a dataflow model. In pipeline terminology, pipeline stages correspond to workflow tasks while instructions correspond to messages. Thus, with a simple linear workflow it will take

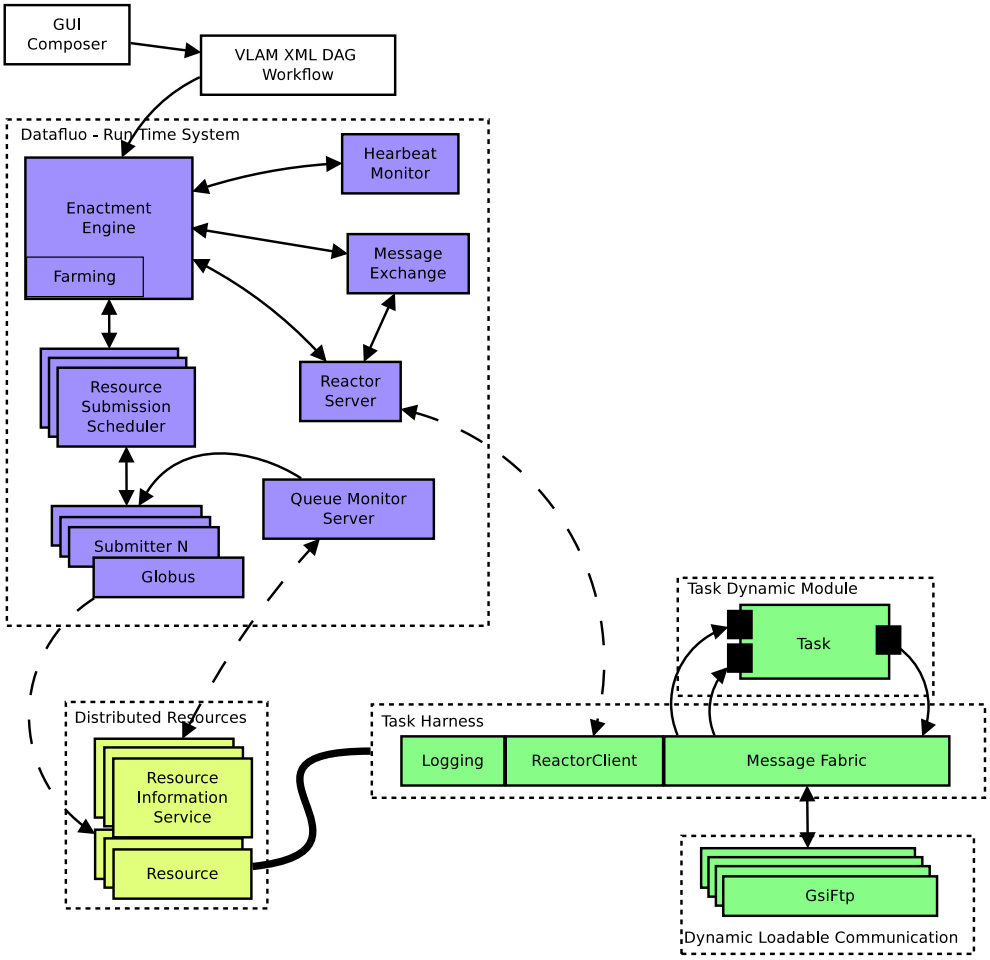


Figure 3.1: Datafluo Run Time System: illustrates the main components making up the server side of the Datafluo architecture. The main component is the enactment engine. Task Harness: is the workflow module harness which get submitted to the resource and hosts the module. Task Dynamic Module: is the actual component where the scientific logic resides. The components are pluggable into the harness. Dynamic Loadable Communication: are the communication plug-in libraries which allow message data to be communicated in different ways for example to a gsiftp server. Distributed Resources: are the actual resources on which tasks execute. The resource information service publishes queue loads to the Datafluo server.

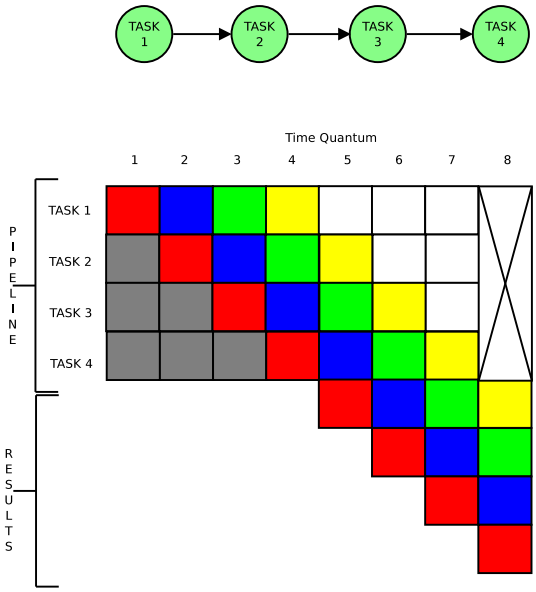


Figure 3.2: Simple linear pipeline stage demonstration. The pipeline depicts a simple 4 task linear workflow. The coloured boxes depict the flow of messages from one stage to the next while the grey and white boxes show idling times.

$n-1$ messages communication to fill up the pipe where n is the number of tasks in the workflow. If each message takes time t to process, the simple workflow will take $(n-1)t$ time before all tasks are processing data. Similarly it will take $(n-1)t$ time to drain the pipe. In the current architecture the pipe startup cost, depicted as the grey boxes in figure 3.2, is an added overhead since tasks have to wait for data to propagate down the pipe. Tasks at the end of the workflow graph have to wait a considerable amount of time before data is received as each task will, typically, have to wait $(k-1)t$ time where k is the task position in the pipe. The pipe drain phase depicted by the white boxes in figure 3.2 is the opposite to the startup phase. It can incur overhead in cases where completed workflow components have to stay alive until dependant components terminate. This might be the case when communication channels between components need to stay active until all parties terminate. As with instruction pipelining in processor architectures as a way to increase throughput, workflow pipelining overlaps task computation thus reducing the workflow makespan. Since all workflow components take different times to process messages, it is difficult if not impossible to ensure a *fully pipelined* system as pipeline bubbles are easily created. Bubbles are created when tasks produce messages at a much slower rate than consuming tasks are able to process hence the consumer has to idle between messages waiting for data. This could be lessened, though not fully eliminated, by initiating multiple copies of the producer which would ideally produce multiple messages at a time.

The Datafluo engine eliminates the startup phase idling time for each task by scheduling tasks only when data is available. Drain-phase idling time is also eliminated through our architecture since communication is decoupled in time and hence tasks can safely terminate without message data loss. The dataflow characteristics allows on-demand process network composition since only actors having work to do will start execution. In essence, this characteristic reduces grid resource footprint in two ways: resources are only occupied when work is available, and graph branches can be systematically disabled through selective channel writing within a task. Individual dataflow tasks can act as conditional branching by deciding, on some internal condition, to which port data is written. Once actors fire and materialize into real computation they act as streaming tasks whereby they continue consuming and producing messages until they either decide to exit or all input channels are exhausted. Termination occurs in a cascading effect whereby initial tasks terminate, prompting dependent tasks to terminate once the last message has been consumed. In grids and typical cluster architectures, applications compete for computation time slots hence an idling task will use-up a resource slot while not producing any useful work. This is contrast to a single computer where typical operating systems are able to context switch processes hence idling tasks do not steal computation. Our Datafluo model is resource friendly whereby it only occupies resources when work needs to be done.

3.1.1 Task Farming

The total work to be done in our workflow architecture can be described as a function of the total messages consumed and produced by the workflow components. Messages in a workflow system are generally bound to some computation hence each message consumed has a respective computation time associated with it. In an ideal scenario, all workflow components take the same time to process each message hence tasks would never have a backlog of messages to process. In practice this is never the case since tasks are isolated programs with their own logic complexities thus message backlog and an eventual bottleneck are common scenarios in a workflow pipelined system. By allowing a backlogged task to replicate itself we can circumvent the workflow bottleneck. This task replication is what we call *task farming*.

The enactment engine has built-in support for *task farming* and *parameter sweeps*. These are common methods used to exploit concurrency in embarrassingly parallel applications. Such applications can be split in many tasks with little or no synchronization between them. Typical scenario for farming is the processing

of large data sets, where each identical task can process different parts of the data. Parameter sweeping is a special kind of farming where the identical tasks act on the same data but with different parameter settings. Farming in Datafluo is achieved through cloning, whereby tasks can be cloned as many times as needed. A clone is an almost identical copy of the parent with some subtle differences such as the inability to clone itself and that ports are shared with those of the parent. Clones are not visible outside the scope of the parent task and hence the workflow system as a whole does not know about clones. Each parent manages its own set of clones and hence avoids changing the semantics of the original workflow. Clones share the same logical ports as the parent and so whatever the parent receives can be accessed by its clones. A task is only allowed to have one farmed port. A farmed port is such a port that the data received on the port can be split between the farmed task pool. Having more than one port would create a data consistency problem. If we have a task with two ports designated as farmed, port 1 has queued a set of messages $(X_1...X_n)$ and port 2 has the set $(Y_1...Y_n)$ then we want to guarantee that a clone that receives message X_1 on port 1 will also receive Y_1 on port 2 since there might exist a casual dependency between both messages. This is not feasible as it incurs excessive synchronisation overhead and can be circumvented by designing a task where ports 1 and 2 are merged into one port and messages X_n and Y_n are merged into one message Z_n. Farming in Datafluo supports three different strategies:

- **Auto-Farming:** In auto-farming, the parent task within the enactment engine is responsible for initiating clones depending on the apparent task load. This is done by gauging the tasks' farmed input port against a pre-set threshold. Since each message is associated with a computation, the *Message Interval Time (MTT)* of a port (i.e. the time taken between message requests) is an approximate of how long the task took to process the message and thus the message queue size coupled with the MIT is an approximate load on the port. The port load is calculated on every message read and is a function $\frac{(St_n)L_{n-1}}{2}$ where S is the message queue size, t_n is n^{th} message processing time and L_{n-1} is the previous calculated load. The function calculates the load as the message queue size multiplied by the time quantum between message requests. In order to smooth out spikes in the load graph we factor in the previous calculated load and take the average. This gives an indication on the time needed to process the data and hence also allows to approximately predict how many clones are needed to process the data

within a certain time frame. The auto-farming further emphasises the single farmed-port restriction since with more than one farmed port it would not be clear which port should be chosen and monitored to initiated task duplication. This auto-farming strategy is best suited for large queues with relatively low message processing time since the load is only calculated after the first message has been processed. In the case where the load is high due to high message processing time and a short queue, the system would have to wait until the first message is processed before it starts cloning. This is a weakness in the system and can be alleviated by pre-emptively taking an auto-farming decision before the first message has been processed in cases where the message takes a long time to process.

- **One-to-One:** This type of farming allows for new clones to be submitted for every message received on the farmed port. This type is particularly useful in parameter sweeps, where each parameter is given an associated clone. Through this system, actors become parameter sweep engines as they are responsible for generating parameter messages to be used by clones. This is in contrast to other systems such as Nimrod [11] where the system itself is a parameter engine. Parametrized tasks would typically include one or more data ports. The engine ensures that all clones, no matter when they join the farm, have a consistent identical view of the data. Thus a latecomer sees the exact same data and ordering as others that have already consumed parts of the data stream. The parent must also buffer the data messages indefinitely to guarantee that any new clone gets access to all the data.

- **Fixed-Farming:** This is a static farming type whereby the user decides beforehand on the number of clones that should be initiated for a given task. Whereas a one-to-one model, tasks consume only one message off the shared queue, in a fixed farm tasks typically come back for more messages and continue processing messages until the shared queue is exhausted. This gives rise to farming in a service oriented way where tasks are long lived and continuously consume messages. Such a scenario would be a parameter set of 100 and the user decides to initiate 5 clones whereby each clone would consume 20 parameters.

3.2 Message Exchange

In a typical grid environment tasks have to percolate through a number of scheduling queues before reaching actual resource. This tends to fragment the

workflow as some tasks might get through the queues while others get stuck waiting for their turn. Some grid middleware systems implement advance reservation systems to tackle the co-allocation problem and guarantee that a number of tasks can be scheduled simultaneously. Reservations are known to degrade the system due to increased wait time [20]. The problem is exacerbated when co-allocating many tasks across grid sites, as in the case of a farmed task.

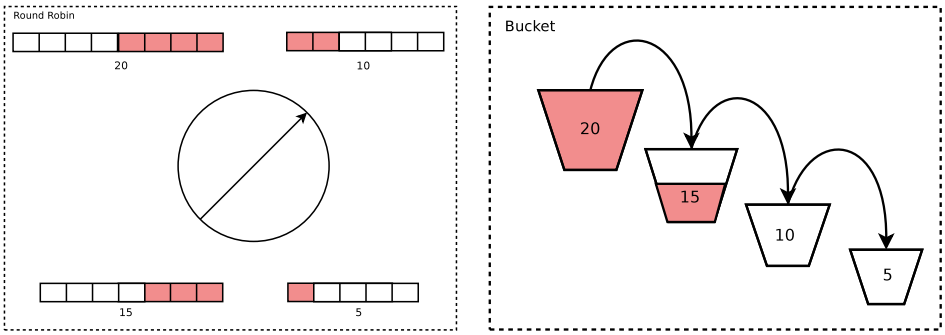
In our Datafluo model we make a weak assumption about task communication. Tasks are allowed to be scheduled out-of-sync and out-of-order, hence communication is not synchronised but buffered which, in essence, decouples task communication in time. This allows tasks to be scheduled without the need for co-allocation. Communication buffering is achieved through the message exchange component. The message exchange binds tasks' input and output ports to message queue and is responsible for routing messages between queues.

The Message exchange aids the enactment engine in achieving the farming capabilities. Clones for a particular task share the same input farm queue and output queues. Every clone picks up and removes the next message in the queue, hence no two clones get the same message. On the other hand, clones having multiple input ports need different treatment since it must be ensured that all clones get exactly the same data. This is achieved by using *shadow queues* on the parent task. When a clone is initiated, its input data queues are attached to the parent shadow queues. The shadow queue acts as a persistent buffer and hence a new clone can grab all the data that has been received by the parent. Termination of a farmed task can only happen once all clones and their parent have terminated.

The message exchange component plays a secondary role in the orchestration of a Datafluo graph. It provides data flow information to the enactment engine, which needs to determine the actor firing events. When a task produces messages on its output ports, the message exchange routes the message to all connected ports which will in turn signal the enactment engine about the new data. The enactment engine will register the data reception on the port. Once a task has data on all its input ports, the enactment engine will proceed to fire the task.

3.3 Job Submission

The architecture exposes a scheduler and submitter interface. The scheduler interface is intended for implementing specialized schedulers whilst the submitter interface is the resource abstraction layer. The scheduler can manage multiple submitters and is able to distribute the workflow on multiple resources.



(a) Round Robin Scheduler (b) Bucket Scheduler

Figure 3.3: Two simple scheduler implemented for the Datafluo architectures. In *(a)* the scheduler fills unequal queues evenly by calculating the fill ratio. In this case 4 queues with 20, 15, 10, and 5 free slots are filled at a ratio of 4:3:2:1 which fills the queues evenly. In *(b)* the scheduler focuses on locality. the free slots are represented as buckets and the scheduler fills the largest bucket first and overflow the extra load onto the next largest bucket.

A simple scheduler setup is a Round Robin scheduler with a number of cluster submitters. This scheduler aims at load balancing tasks across multiple cluster sites by submitting tasks to submitters in a circular fashion. The Round Robin scheduler takes into consideration different sized queues hence three queues having 15, 10, 5 free slots will have a submission ratio of 3:2:1 respectively. That is, the first queue with 15 slots will get 3 tasks, the second with 10 free slots will get 2 tasks and the last queue will get 1 task. This ratio ensures the queues fill up at the same rate. The ratio is calculated by selecting the smallest queue and dividing all the queues by the smallest queue size.

A second scheduler is the bucket scheduler. This scheduler aims at optimising locality by grouping as many tasks as possible onto one resource. The submitters are ordered according to their queue size. The available queue slots represents the submitters' bucket. The scheduler starts by filling up the largest bucket and when full, it overflows tasks to the second largest bucket. Once slots become available in a larger bucket, the scheduler tries to fill it up again by scheduling next tasks into this bucket. Optimising task locality aims at improving communication since tasks are geared to produce data on the closest available data storage.

Although the above scheduler take automatic decisions, one can imagine a scenario where a user wants to manipulate the resource priorities at runtime thus disabling a resource completely or giving it higher priority. Such decisions might allow a user to include a Cloud resource (which has an associated monetary cost) at runtime when the other resources become clogged. For such a scenario we have

an interactive scheduler where the user can set priorities for each resource. Before every submission the resources are sorted and the highest priority resource with available slots is chosen.

Schedulers such as the round robin and bucket scheduler need resource information so as to make a better decision. The queue monitor server component is responsible for listening for queue statistics such as queue size and queue free slots. Resource information services on the resources such as cluster head node send queue information to the server. The queue monitor will update the submitters' data which in turn can be used by the schedulers to take an informed decision.

Submitters can be of any kind such as *Sun grid Engine (SGE)*, Globus, Gminion, and local. Submitters are entrusted with setting up all necessary configurations for the actual submission which may include staging in/out files and redirecting stout/stderr. Tasks can be bound to a particular submitter through parameter settings hence forcing the scheduler to submit the task on the specified resource.

3.4 Reactor Server

All communication between the tasks and Datafluo is handled through a command server which accepts commands for checking mail, posting mail, and heartbeat messages. The server works in a passive mode and never initiates a connection to a task. This is due to the inherent inbound communication restriction between cluster sites. Tasks are responsible for pulling message and pushing new messages. So, as not to inundate the server with check mail requests, tasks implement an exponential back-off $\frac{e^n}{(n+1)}$ where n is the check mail attempt counter. The back-off is capped at a threshold and reset to 0 when a new message is received.

3.5 Heartbeat Monitor

The heartbeat monitor is a simple mechanism to detect unexpected task failures and issue resubmissions. The monitor iterates through the workflow task list at pre-set intervals and checks the last received heartbeat for every task. A task is resubmitted if it abruptly stops sending heartbeats. In case of resubmission, tasks are resubmitted to the same submitter hence bypassing the scheduling algorithms.

3.6 Task Harness

The second part of the architecture depicted in figure 3.1 illustrates a 2-input 1-output port task as, the executable submitted to the resources. The architecture is based on a plug-in model whereby the task and communication modules are dynamically loadable into a harness program. The harness has three main parts: a logging system, a reactor client, and the message fabric. The crux of the harness is the message fabric which binds loadable communication libraries to the task input/output ports through a system of message queues. The message fabric sets up message queues for every port on the task module. These queues are bound to queues exposed by the communication library. The communication libraries implement user defined protocols. A user defined protocol such as `gsiftp` would read and write binary message data to `gsiftp` servers while `pgsiftp` would get a file from a `gsiftp` server and pass a pointer to the local file up to the task. The `pgsiftp` library is useful for legacy and third party software that expect files as input as opposed to raw binary data. When tasks are initialised on the resource they first contact the reactor server for configurations. The configuration includes the ports setup and the list of available servers for publishing messages. The task selects the best server depending on the `ping` round trip time. Part of the configuration includes the protocols available for a particular server (e.g.`gsiftp`). This information would be used to load the appropriate communication library. The messaging fabric allows component developers to focus on the scientific logic as communication is hidden behind the harness.

3.7 Concurrency

Having described the whole system we can point out three levels of concurrency achieved through this architecture: task level concurrency, pipeline concurrency, and farming concurrency. Task level concurrency is achieved through the inherent nature of dataflow process networks as multiple tasks can fire simultaneously. Computational overlap between pipelined tasks achieves a finer level of concurrency dependent on the frequency of message passing between tasks. The higher the message frequency the greater the computational overlap while lower frequencies tend to produce more pipeline bubbles. Farming achieves data concurrency between independent tasks by splitting data n-wise between n identical tasks running concurrently.

Chapter 4

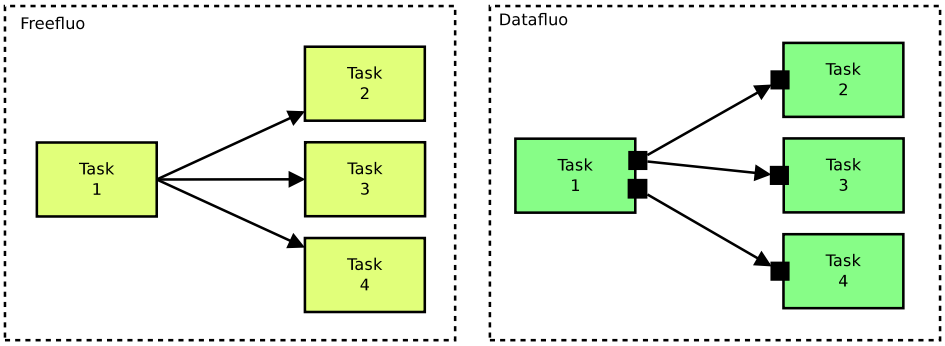
Datafluo Implementation

The following describes implementation issues regarding the described architecture. The implementation is split into two main sections: the server side, and the client harness side. The server side implementation is written in Java. The main objects depicted in figure 3.1 map to the main Java packages. The architecture objects are mostly decoupled and can be easily swapped out for other implementations.

4.1 Enactment Engine

The Datafluo enactment engine is based on the Freefluo [2] engine. As described in section 2.7, Freefluo is a flexible Java workflow orchestration tool aimed at web services. Freefluo orchestrates a workflow in a DAG fashion, that is, it will only fire tasks once parent tasks have been completed and not when parent tasks produce data. Thus Freefluo has no notion of a task port and only represents dependencies between tasks without defining what kind of dependency exists between tasks.

In our prototype we extended the Freefluo implementation to model our dataflow model. A new port object was included to model actors having ports.



(a) Freefluo workflow (b) Datafluo workflow

Figure 4.1: Freefluo workflow representation vs the Datafluo workflow representation which includes ports which distinguish between dependencies.

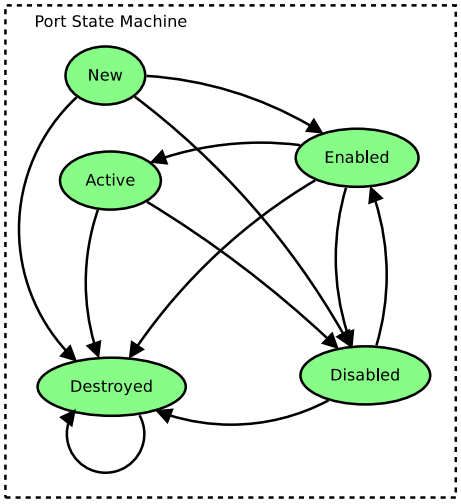


Figure 4.2: Datafluo port state machine. An *Active* state denotes that a port has sent or received at least one message. The port remains in an Active state until its either destroyed or disabled.

Figure4.1 depicts the difference between the Freefluo workflow representation in 4.1(a) and a possible Datafluo similar representation in 4.1(b). In Freefluo, tasks 2, 3, 4 all depend the same way on task 1 hence once task 1 terminates all other tasks commence. In the Datafluo example, task 2 and 3 depend on port 1 of task 1 while task 4 depends on port 2 of task 1 therefore once task 1 produces data on port 1, task 2 and 3 start simultaneously while task 4 still waits for data to be produced on port 2.

Tasks have a list of input ports and output ports. Dependencies are created between ports and not between tasks. A port is a state machine and is illustrated in figure 4.2. This state machine is implemented in a similar fashion to the task and flow state machines described in section 2.7. When the first message is received by a port, its state is changed from *Enabled* to *Active*. Tasks listen for port state changes and will fire when all input ports have been elected to the *Active* state. A port becomes *Actvie* on the first message and remains so until it is destroyed at the end of the message stream.

The state machine for the flow, task, and port objects are implemented in a similar way. The objects are composed of three main objects. One is the object itself such as `Port`. The second object is the abstract state object such as `PortState` which all other state (*New, Enabled, Active, Disabled, Destroyed*) must implement and the third is the state event object which carries the event data. The event object carries important information such as the source object and the new state. The port object implements the port logic which includes managing

an array of connections to other ports, setting the direction of a port (a port is unidirectional so it can only be set to input or output), and managing the array of state listeners. Any object interested in the port state changes must register itself to the port through `addPortStateListener`. This ensures the any changes to the port propagates to the registered listeners. One important port listener is the task to whom the port belongs. The port (task and flow alike) includes a `PortState` object which implements the state transitions. A state object is an abstract object which defines the possible state transitions as methods though it does not implement any of the methods. Each state object must implement the abstract state class and implement the relevant state transitions hence the *New* state will implement the *Enable*, *Disable*, *Destroy* transitions since those are the only allowed transitions from a *New* state. If any other transition is made from the *New* state it will try to execute the abstract state method which will throw an `IllegalStateException`. Every state transition within a state object performs the same basic functionality which includes changing the internal state and calling a callback method (e.g.`portStateChanged`) on the relevant object which passes the state event object in this case `PortStateChangedEvent`. This callback will in turn call the callback methods on all the registered state listeners.

The Datafluo enactment engine implements several Freefluo interface classes namely, the `Engine`, `WorkflowParser`, `WorkflowInstance`, `Task`, and `Port`. The new implemented objects implement Datafluo specific logic. The `VlamEngine` which inherits the Freefluo `Engine` overrides the `compile` method which takes the WS-VLAM XML graph. This calls the `VlamWFParser` which is responsible for parsing the actual XML file and generate a Datafluo representation. The parser will first create a list of `VlamDatafluoTask` and then proceeds to generate a list of `VlamDatafluoPort`. A `VlamDatafluoTask` is given a Universally Unique IDentifier (UUID) and parameters such as farming and host are extracted from the XML file and set in the task object. The port objects are iteratively connected to each other so as to represent the networked graph. Whilst generating the port connects, message queue objects are also created for each port. A `VlamDatafluoPort` extend the Freefluo `Port` and adds a message queue object associated with the port and a `queueRaisedEvent` method which listens on the message queue object for messages being received or consumed.

The `VlamDatafluoTask` extends the Freefluo `Task` and adds quite some functionality. The `VlamDatafluoTask` overrides the `handleRun` method which is called by Freefluo when a task is ready to fire. The new `handleRun` will basically add the task on the Datafluo's scheduler runnable queue through `add-`

`RunnableTask` . This will be picked up by the designated scheduler and submitted using one of the selected submitters. Other important functionality with the `VlamDatafluoTask` object is the cloning mechanism. The task listens on the port for messages. When a message is received the `portReceivedMessage` is called and can take 3 possible actions. If the port on which a message has been received is earmarked for farming then the task can be cloned if the conditions for fixed-farming or one-to-one-farming are met. The fixed-farming will create a fixed number of clones which would have been defined by the user beforehand and then farming is disabled for the port so no further cloning can take place. With one-to-one-farming, a new clone is created for every message that has been received. If the message belongs to a normal port, that is not farmed, no further action is taken. Cloning is also restricted to tasks that do not have any dangling clones. A dangling clone is one which has been submitted but has not yet reported in as alive.

Cloned tasks are identical to the base or parent class with some important differences. A clone has a new UUID, the cloned task cannot clone itself, the designated farmed port queue and output port queues are shared amongst parent and all clones while new queues are created for input data ports (data ports are those ports that are not farmed so messages on the parent data port must be replicated on the data ports of all clones). The latter procedure is needed since all clones must receive the same data on the data input ports as the parent which would not be the case with shared ports since messages read by one clone is removed and not accessible by any other clone. Furthermore clones may join the farm at a later stage when the parent has already consumed messages from the data ports and hence the clone would not see all the data but a part of it. To guarantee data consistency across all clones, the parent sets up shadow queues as depicted in figure 4.3 for each of its data input queue (not famed queue). The shadow queue acts as a persistent buffer for messages being received. The parent can consume messages normally from its live data input queue. Once a clone is made, new input queues are setup for the clone and attached to the parents shadow queues. This allows the clone to receive all the messages that have been received on the parent queue irrelevant of the time the clone joined the farm.

All input ports irrespective of cloned tasks or not have an associated reserve port. This port keeps track of the last message consumed by the port. Datafluo will re-insert this last message at the end of the queue in the eventuality that a task has crashed and is re-submitted. The reserve port ensures that the messages consumed during a task crash can be replayed. All clones and parent output

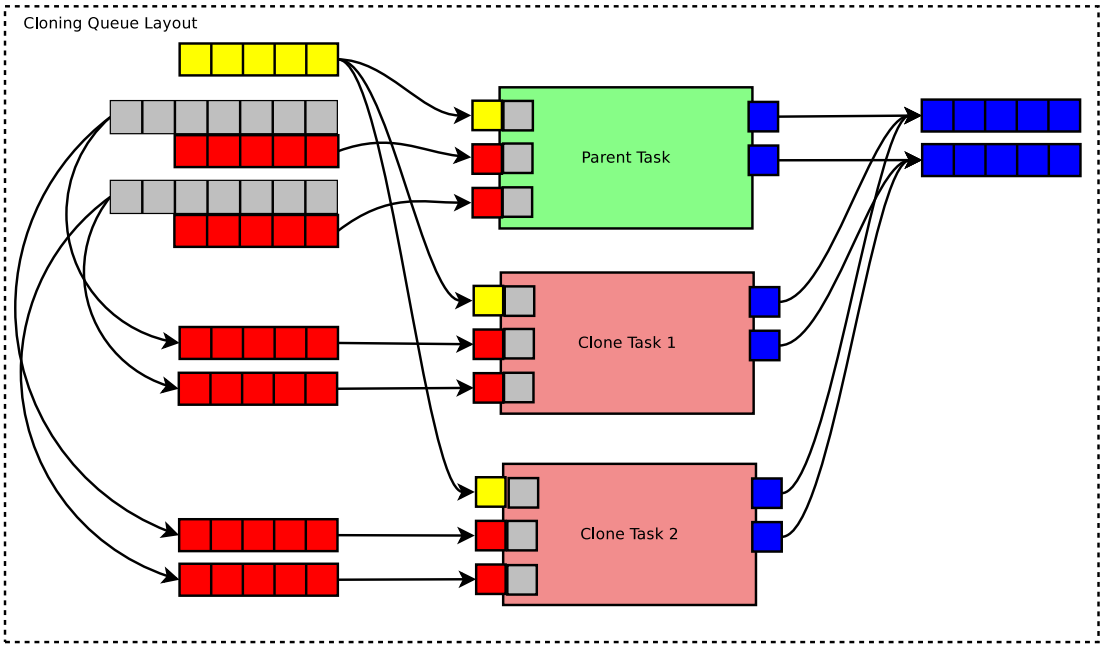


Figure 4.3: The figure illustrates the cloning queue strategy. The image show a parent task and two clones. The tasks have 3 input ports one of which is a farmed port (yellow) and 2 output ports (blue). The yellow parts illustrate the farmed port and its associated queue. The red parts illustrate the input ports and their queues while the grey queues depict the shadow input queues. The blue parts show the output ports and their queues. All the clones including the parent share the same farmed port queue and output queues while data input queues (red) are attached to shadow queues (grey) on the parent task. The shadow queues act as persistent buffers for the clones. All input ports have an associated reserve port which always keep the last message consumed. This ensures that no messages are lost in case of resubmission.

messages to the same output queue. This might cause data hazard issues with application sensitive to message ordering such as a *mean-shift tracker* [1]. The solution to this problem, although not implemented, lies in ordering messages according to the message header sequence number. Typically the first messages to enter the workflow are given a sequence number. This number could be tracked through the system to order any new messages generated on the bases of the input message so any messages generated by processing input message 1 should be queued before any new messages generated by processing input message 2.

As for task completion, a task cannot be safely marked as completed before all clones have terminated. For this reason a task never explicitly completes but tries to complete through the `tryComplete`. The `tryComplete` method first checks if the terminating task is a clone and, if so, it will simply remove the clone from the parent's clone queue. If the terminating clone was the last clone and the parent is already in a terminating state then the parent can proceed to terminate successfully. If the terminating task happens to be the parent then, if there are no clones, the parent will just terminate. If clones still need to terminate then the parent will signal it has terminated without actually completing and wait until all clones have terminated. This procedure ensures that all clones *join* on the parent task.

Auto-farming occurs on a consumed message as opposed to a received message. Every time the task pulls a message from its input queues the `portConsumedMessage` method is called. This determines if the task should be farmed or not. Auto farming works within a set of limits set beforehand. These thresholds are in place so as not to allow over farming of the task. The `loadThreshold` is a fixed threshold which is compared to the load function result and would not farm unless the threshold has been met. The time between cloning the task and the actual execution of the tasks is a grey zone since during this time the load remains high but clones have already been sent out. To limit cloning during this grey zone we only allow cloning on tasks that do not have dangling clones. The result of the load calculation which predicts the number of clones needed to satisfy the `loadThreshold` may be too high for the resources to handle. For this reason we define the `maxCloneBurst` threshold which limits each cloning attempt to this threshold. If after submitting the clones the load remains high, auto farming can opt to resubmit a further clone pool. This procedure continues until the apparent load in the farmed input port remains below the `loadThreshold` limit.

[1] A system for tracking objects in an video stream

4.2 Message Exchange

The message exchange core classes implement the actual message routing between tasks an allow the enactment engine to orchestrate the workflow graph. The core logic of the message exchange is implemented in the `MessageExchange` object which exposes methods to create new queues through `createMessageQueue` and create links between queues through `createLink` . The actual message routing takes place within `queueRaisedEvent` which is called whenever a new message is received on a queue. The routing procedure distinguishes between a shadow queue and a normal queue. With a normal queue messages are removed once routed to connected queues while with a shadow queue messages are routed, the queue cursor is advanced and the message is left on the queue. This allows later clones to retrieve all the messages received by the parent. The `Queue` object implements the message queue as a linked list. Each queue is identified by a key which is defined as the combination of the message exchange id, task id, and port id. The `Queue` implements functionality to add and remove messages as well as maintain a shadow queue cursor. The `Queue` also implements the load function by recording the time taken between message retrieval. This time quantum is then used in `calcLoad` to calculate the queue load as function of the queue size and time quantum as described in section 3.1.1. Queue messages are `Message` objects which contain a header and the message data represented as a string. The message exchange exposes queue events through the `IQueueEventListener` whereby interested objects such as `Port` listen for such events. In the case of `Port` object, the port informs its task that a message is received or consumed and hence the task can activate the port which allows the enactment engine to orchestrate the dataflow model by invoking tasks which have all input ports activated. Hence in our architecture the message exchange has a dual role: that of message routing and that of providing events for the enactment engine. One can conceive of other modules that can spoof on the message exchange such as a provenance module which would store the messages in persistent database and later be used for fault tolerance or display the data transformations from start to finish.

4.3 Task Submission

The submission core classes deal with the actual job submission and can be categorised into two main objects: the `Scheduler` and the `ISubmitter` objects. The `Scheduler` is a super class which all schedulers must extend to submit jobs. The scheduler is threaded hence it runs in parallel to the enactment engine and

exposes the `run` method which must be implemented by inheriting schedulers. The basic setup of all schedulers is a vector run-queue which tasks use to add themselves through `addRunnableTask` . The scheduler will then run an infinite loop reading tasks from the run-queue and submitting them according the scheduling algorithm. The run-queue is synchronised through a semaphore which allows the scheduler to block when the run-queue is empty. The queue is also protected through a mutex so that the scheduler and enactment engine do not access the run-queue concurrently as this would otherwise corrupt the queue.

As described in section 3.3, the system implements three different schedulers: the `RoundRobinScheduler`, the `BucketScheduler`, and an `InteractiveScheduler`. All have similar implementation although the logic varies as described in the architecture. The round-robin and bucket schedulers are at best $O(1)$ and at worst $O(n)$ as the worst case the schedulers need to iterate through the list of submitters to find a suitable one. The interactive scheduler depends on the Java inbuilt sorting complexity which, for small lists, uses insertion sort which is $O(n^2)$ at worst. Thus the worst case for the interactive scheduler is $O(n^3)$. The schedulers depend on resource information from the submitters and will initially block until all submitters report their queue statistics which includes the queue size and their free slots. Once all information has been gathered the submitters are ordered according the scheduler's logic and the best submitter is chosen for the task. The tasks can bypass the scheduling mechanism by setting their `host` variable which is then used to bind the tasks to a specific resource.

The actual task submission is done through one of the submitters. All submitters must implement the `ISbumitter` interface which exposes methods such as `submit` and other methods related to the the queue information such as `getAvailableSlots` . Since submitters can take some time to actually submit the task, submitters are threaded. The SGE DAS3 submitter is done through the `SimpleGramSubmitter` and `SimpleGramSubmitterThread` which basically builds a *Resource Specification Language (RSL)* XML file that includes the harness executable location, the server IP and port to contact, the task UUID, the `stderr/stdout` file redirections, and the security credentials. The RSL file is then submitted using the `globusrun-ws` command which submits the job to a specified resource chosen by the scheduler.

Most schedulers would require some form of resource load statistics to take a better scheduling decision. For this reason, the submission core includes a passive server that listens for resource loads such as the queue size and available free slots. This is done through the `QueueMonitorServer` and its worker thread `Queue-`

MonitorWorker. The server will match the submitter with the telemetric data received and update the submitters' slots and freeSlots variables which can be accessed by the schedulers so as to order the submitters.

As part of the submission core, the architecture also includes a heartbeat monitor which is implemented in HeartbeatMonitor object. The heartbeat monitor is a threaded infinite loop which iterates over all tasks at intervals of 60 seconds and checks each tasks' last heartbeat through the getLastHeartBeat method. If the time difference between the last heartbeat is beyond the threshold, the task is resubmitted. Resubmission involves queueing the task back onto the run-queue.

4.4 Reactor Server

As described in section 3.4, the architecture implements a command server, the ReactorServer. The server decouples all network traffic from the Datafluo core objects. It is a threaded server which, by default, listens on port 5555. The server is threaded hence running concurrently to other Datafluo core threads. Since each connection opens a new thread and each task can open many connections during its lifetime, the server resources can be easily exhausted by opening many threads at once. For this reason threads are managed through a fixed thread pool and are recycled once a connection has terminated. The thread logic is spread across two classes: the WorkerRunnable, and CommandHandler. The WorkerRunnable implements the server threads and acts as a switch for the incoming commands. The first byte of each byte-stream denotes the command type. The WorkerRunnable reads the first byte of the stream and switches accordingly. Control is passed to the CommandHandler which handles the commands and generates a response where applicable. Commands include:

- CHECK_MAIL (0x7E): The command string includes the queue ID which is associated to a port. The CommandHandler looks up the queue using the ID. The handler can take 3 possible actions: if the queue has messages it will send the next message and reply with a SENDING_MAIL command, if the queue is empty it will just reply with NO_MAIL command, and if the port has been destroyed by the task, the handler replies by PORT_DESTROYED command. A destroyed port may still contain queued messages hence the PORT_DESTROYED is only sent once the queue is empty.

- POST_MAIL (0x7D): In a post mail command the stream is read and queue ID is extracted, the queue is looked-up and the message is added to the queue. The message exchange will then handle the message.

- `GET_CONFIG (0x7B)`: Submitted tasks need to ask for configurations upon initialisation. This is done through the `GET_CONFIG` command. The configuration contains 3 main types of configuration entries: `Module` which include module configuration such as module name and parameters, `Queue` which contains queue to port binding information such as queue ID, port name, port direction, and parameters, and `Server` which contains a list of servers used for storing intermediate message data.

- `HEART_BEAT (0x7A)`: The heartbeat command handler calls the `heartBeat` method on the task which updates the `lastHeartBeat` variable.

- `COMPLETE (0x70)`: This command signals the end of a task and calls the tasks' `tryComplete` method.

The server and the harness task do not attempt to keep the connection alive hence once a command has been served the connection is severed from both sides. Although opening a new connection every time is a communication overhead, the architecture assumes the resources are unstable and can not rely on stable open connections. Furthermore the open connections would limit the number of tasks the server can handle. Some optimisations can be implemented to reduce the communication overhead by allowing the connection to remain open for a short period of time hence reusing the connection during burst communication.

4.5 Task Harness

The task harness implementation revolves around the idea of a pluggable architecture which as described in section 3.6 enables the complete decoupling of the scientific logic from the rest of the system including the harness itself and the underlying communication libraries. The harness system is implemented in C/C++ using extensively the *Standard Template Library (STL)* for data structures such as vectors and lists. The pluggable components communicate between each other using a system of message queues which have nothing to do with the messaging done by the server. The implementation is mainly split up into 5 parts: the message queue (`MessageQueue`), the reactor client (`ReactorClient`), the message fabric (`vlport2`), the communication pluggable modules and the scientific pluggable workflow module.

The `MessageQueue` is the binder for the whole system. It basically binds the workflow module to the harness and the communication libraries to the harness

as well. The object implements a list as its queue structure and exposes functionality to manipulate the list. Apart from the obvious `Read` and `Write` functions, the message queue implements two synchronisation functions: `Signal`, and `Wait`. These functions act as a semaphore and are used to synchronise the consumer and producer. The `Read` and `Write` have separate synchronisation to guard the actual queue structure from concurrent access. The `MessageQueue` also carries a state variable which is primarily used to signal events between the producer and consumer. A typical usage of the state is by the consumer to alert the producer that it will not read any more messages and hence its safe to abandon the queue.

The `ReactorClient` is responsible for communicating with the Datafluo server. The commands flowing to and from the server are described in section 4.4. The functions exposed through the `ReactorClient` allow the harness to communicate to the Datafluo engine through functions such as `CheckMail`, `PostMail`, and `SendComplete`.

The scientific logic is captured with the workflow module part of the harness. The modules are self contained and pluggable, that is they can be programmed outside the scope of the harness whilst the harness code can change and evolve whilst the modules need not be recompiled. This allows flexibility in the programming paradigm. All modules have to follow the `IModule` interface and implement the module's constructor, destructor and three virtual functions which will be called by the harness. These functions are: `init` which gets called after the constructor to perform initialisation routines such as loading parameters that are passed as arguments to the `init` function, the `start` function which is the actual start of the scientific execution, and `stop` which implements implements termination routines before the destructor. The module constructor is where the ports are created and special environment variables can be set. Program 4.2 shows a typical template implementation which simulates a two-input, one-output workflow module. The constructor initiates the ports and maps the input (RX) and output (TX) ports to a name (lines 7-9). The name is used by the harness to bind the port to a queue at the message exchange. An optional `init` function (line 12) can be implemented to parse any passed parameters. In this template example, the `start` function at line 15 implements an infinite loop to read the input ports sequentially at lines 17-18. `READ_PORT` is a macro which returns the next message on the queue identified by its index number. If any of the messages are NULL, the loop will exit and the whole module exits. After successfully reading input messages from both ports, the scientific logic replaces line 23 where the messages are processed and eventually some data is ready for transmission. At line 25 a new message is created an

the data length and reference to the data is set. The output data is finally sent out of the module through `WRITE_PORT`. The message will be picked up by the harness and taken care of. The module must signal the harness to retrieve the next messages for the input ports. This is done with `SIGNAL_RX_PORT` which takes the port index as an argument. Finally once the loop exists, a NULL message is written to all output ports hence signalling the harness that the ports have been closed.

In the template implementation, the scientific programmer does not need to handle underlying communication, data is read and written to queues which adequately abstracts the underlying system of getting messages between components. The `REGISTER_MODULE` at line 38 is the self registration macro[49] shown in program 4.1 which gets called for when the compiled object is dynamically loaded. The macro consists of a factory function which returns an instance of the `Template` and a proxy class whereby the constructor of the proxy class inserts a pointer to the factory function (line 7) in a shared hash table, `gModuleFactory`, between the harness and the module.

Program 4.1 Module Self Registration

```
1  #define REGISTER_MODULE(NAME)
2    extern "C" {
3    IModule *maker(){   return new NAME; }
4    class proxy {
5      public:
6          proxy() {
7            gModuleFactory[#NAME] = maker;
8          }
9    };
10   proxy p;
11   }
```

The implementation of communication libraries work on the same principle as the workflow modules. The libraries have to follow an `IComm` interface. The main difference between a workflow module template and a communication template is the fact the a communication module will only implement one queue. Similar to a workflow module, the communication module also has an `init`, a `start` and a `stop` function as well as the constructor and destructor. The communication modules are split up into two separate libraries for each protocol: one that handles the input and the other for handling output so for the `gsiftp` protocol one would expect two libraries: `InputGsiFtp` and `OutPutGsiFtp`. Upon calling the `start` on an input module, the module would typically read the parameters which would include a file URL and host-name in the case of a `gsiftp` protocol. The module would continue

Program 4.2 Workflow Module Template

```
1  #include "IModule.h"
2
3  class Template : public IModule {
4     public:
5       Template(){
6         INIT_PORTS();
7         MAP_RX_PORT(1,input_1);
8         MAP_RX_PORT(2,input_2);
9         MAP_TX_PORT(1,output);
10      }
11      ~Template(){}
12      void init(vector<string>* rParam){
13        //Read rParams
14      }
15      void start() {
16        while(1){
17          MessageQueue::Message* im1 = READ_PORT(1);
18          MessageQueue::Message* im2 = READ_PORT(2);
19
20          if((im1 == NULL) || (im2  == NULL))
21            break;
22
23          //Process im1 and im2
24
25          MessageQueue::Message* om1 = new MessageQueue::Message();
26          om1->mDataLength = [data length];
27          om1->mpData = [pointer to data];
28
29          WRITE_PORT(1,om1);
30          SIGNAL_RX_PORT(1);
31          SIGNAL_RX_PORT(2);
32        }
33          WRITE_PORT(1,NULL);
34      }
35      void stop(){}
36 };
37
38 REGISTER_MODULE(Template);
```

and retrieve the file from the server using `globus-url-copy`. The file is read and a message is created with the file data being the message data. The message is queued on the local module queue which is later picked up by the harness and swung onto the workflow module's bound input queue. On the other hand, the output communication module would immediately block on its local queue waiting for messages to arrive. Once a message is retrieved (in the case of `gsiftp`), the module will extract the message data onto a local file and copy it to the GsiFtp server. As with the workflow modules, the communication modules are also self registering and are registered inside the harness in a hash table `gCommFactory`. Although we have created communication libraries to handle the `gsiftp` protocol, communication can take any form. In fact we have written pipe, TCP/IP and file communication libraries for debugging purposes that do not use intermediary servers. Although our Datafluo relies on messages to orchestrate the workflow, one could conceive a communication library that works on a peer-2-peer fashion for frequently communicating tasks on a cluster. In this case, Datafluo messages would merely contain configuration messages as how to setup the network. If we have two tasks that want to communicate directly then we could have this typical scenario: the Datafluo engine submits the first task which generates an output message containing its IP and port. After receiving the message on the exchange, Datafluo will fire the next task which will read the configuration message and setup a connection directly to the other task. In this case Datafluo would be unaware about the communication going on behind the scene and hence cannot provide any support.

The crux of the implementation lies within the `vlport2` which we refer to as the harness. During initialisation, the harness first sets up the working environment such as creating working directories. It will then proceed to contact the Datafluo reactor server for configurations using the `GET_CONFIG` command. This configuration contains information about the scientific module to load, the queues and available data servers. The harness also initiates the heartbeat thread which sends a heartbeat every 30 to 45 seconds. The 15 second discrepancy is so to eliminate the possibility that many tasks send a heartbeat simultaneously although this would still be unlikely since other circumstances such as queue wait times would have already offset tasks significantly. The last 15 seconds are calculated randomly.

The scientific workflow module is loaded through the `module_startup` function. Using the configuration information, the harness tries to locate and load the *dynamically linked shared object library (SO)*. Upon loading the SO file, the

module auto registers itself to a module hash-table in the harness whereby the key is the module name itself. Thus the harness can then create an instance of the module object. Since all workflow modules must conform to an interface virtual class, the harness knows what functions can be called on the module. Using the parameters extracted from the configuration, the harness calls the module's `init` function passing the parameters as arguments. The module's `init` function would be typically implemented by a scientific programmer and would include module specific environment setup such as environment variables and module port setup. Once the module is loaded, the harness can access its public variables and functions thus it will iterate through the module's ports (message-queues) and register references to them within the harness data structures. The actual scientific computation starts once the module's `start` function is called. This is done by the harness on a separate thread hence the harness and the actual scientific execution are executed concurrently.

After the module has started, the harness proceeds to initialise and bind the module's ports to the message queues on the Datafluo server side. This is done through name binding where the name of the port must match the queue name on the server. For each input port, a thread, `mailbox_listener`, is initiated. This is responsible for handling the module's input. The `mailbox_listener` thread is an infinite loop with the following steps: the `ReactorServer` is contacted to check for messages, if no messages are available the thread will sleep. The sleep time increases exponentially as described in section 3.6. If a message is retrieved, the message data is parsed and the protocol extracted (e.g. `gsiftp`). The harness looks up the appropriate library to deal with the protocol. The library is loaded and the message is tokenised and passed as parameters to the communication module. The thread will block waiting on the communication module's queue. Once the communication module writes data to its queue, the harness picks up the message and swings it to the module's bound input queue. The harness will then block on the queue using `Wait` until the workflow module signals (`Signal`) that the next message can be retrieved from the Datafluo server. This synchronisation is needed since we do not want the harness to retrieve all the messages instantaneously from the server as this will make load balancing futile. Furthermore, the module may decide to exit and hence the retrieved messages that have not yet been consumed by the module would be lost. If a `PORT_DESTROYED` reply is retrieved from the server the module is notified of the port's destruction and the infinite loop is exited which also exits the `mailbox_listener` thread.

Whilst the `mailbox_listener` deals with handling the module's input, the

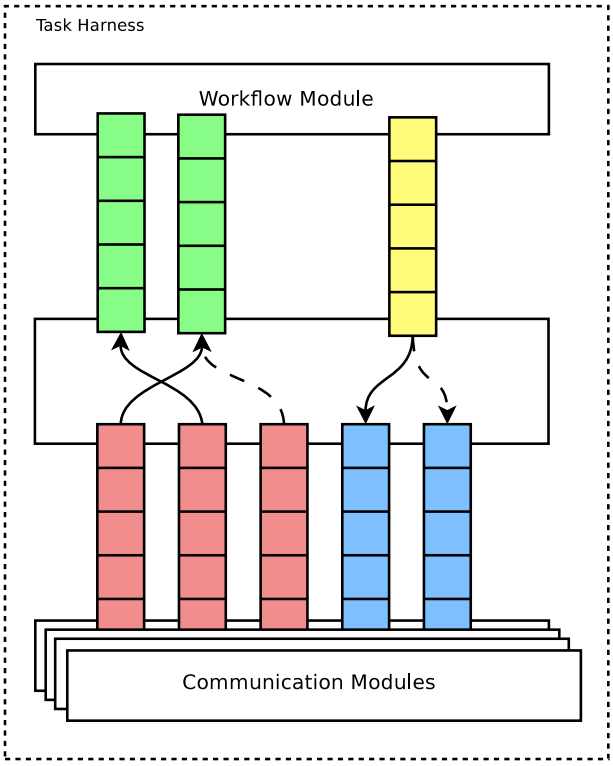


Figure 4.4: Binding the workflow module ports to the communication libraries through message queues. The example depicted here consists of a workflow module with two input queues (green), one output queue (yellow), and five communication queues three of which (red) are input queues. Communication modules put messages on the input queues (red) and pull messages from the output queues (blue). The message fabric routes messages between the modules queues and the communication libraries queues.

`module_output_listener` is the thread that deals with module's output. Before outputting any data, the harness has to decide which server is best for communication this is done through `get_best_server` which iterates the server list and finds the one with the least ping time. The `module_ouput_listener` thread blocks on a queue `Read` waiting for the module to write data on its output port. Once data has been written, the harness will proceed to send the data by checking the protocols supported by the server and choosing the appropriate library for communication. Each server entry in the configuration contains the supported protocol (e.g gsiftp or ftp). This information is used by the harness to load the communication library. The library is loaded in the same fashion as with the input communication libraries. The message is written on the communication queue for subsequent transmission. A mail message is then constructed and sent as a `POST_MAIL` command to the Datafluo message exchange through the `ReactorServer`. The module can signal the task harness that its output has terminated by writing a NULL message to it output queue. This will force the `module_ouput_listener` to break out of the loop and exit the thread.

Chapter 5

Applications and Results

For the purpose of testing our Datafluo architecture we made use of three different applications which took the system through its paces. Two of the applications: *SigWin-detector* and *Wave* have been ported from the current WS-VLAM architecture while a third *HistogramDifference* is a synthetic application built from scratch tailored at exploiting the various features implemented by the Datafluo system.

The following sections describe the applications used and show various results obtained from running the applications within the Datafluo system. Our test bed is the DAS3 cluster which is composed of 5 clusters located at: University of Amsterdam (UvA), Multimedia lab in UvA (Mult-UvA), Vrije University (VU), Technical University in Delft (TUDelft), and Leiden Institute of Advanced Computer Science (Liacs). At the time of writing, the clusters where composed of: 28 nodes for UvA, 41 nodes for Mult-UvA, 79 nodes for VU, 64 nodes for TUDelft, and 23 nodes at Liacs (235 in all). We rarely had absolute access to all the cluster nodes since the clusters are used by other researchers at the different institutes hence our results are relative to the resource load at the time the tests were run. Each cluster has a head node which also acts as a GridFtp file server. Jobs are submitted using the Globus which is installed on all the head nodes. Globus, in turn, uses the internal Sun Grid Engine to submit jobs to the cluster nodes. With Globus we are able to submit cross cluster jobs while also delegating the necessary credentials so that tasks can use grid resources such as the GridFtp servers. For our testing we ran the Datafluo engine on the UvA head node `fs2.das3.science.uva.nl` which then distributed the jobs to other clusters.

So as to illustrate the Datafluo enactment engine in work we illustrate three types of graphs: a resource load graph, a module graph and cluster distribution graph. The resource load graph shows the real load on the resources during our execution which also includes the load imposed by other users. The load is a percentage of the used slots with regards to the actual available slots hence a load of 100% means the resource is full, and a load which is greater than 100%

means the resource has queued tasks waiting for execution. The module graph shows the actual module execution start time, run time and waiting time. The waiting time is the time between the Datafluo engine event to fire the task and the time the actual task starts executing on the resource and is represented as the thin line preceding the bar on the graph. The actual run time of a task is the colour coded bar where the start of the bar states the start of execution and similarly the end of the bar states the completion. The bars are colour coded to distinguish between different tasks. The cluster distribution graph is an identical copy of the module graph but the colour coding represents the resource on which the task was scheduled. Since the execution time is heavily dependant on the load on the resources at the time of execution, execution time may vary considerably between runs for which case an average runtime by itself does not say much. For this reason we also present the standard deviation for the sample set execution.

5.1 SigWin-detector

The SigWin-detector [36] workflow application stems from the biology camp. The aim of the application is for analysing ordered sequence of values such as gene expression data but can even be used for other data such as local time series of temperature. Data is analysed in SigWin-detector so as to identify regions in the sequence where the median value is higher than would be expected by chance if data ordering is not relevant [40]. In gene expressions it identifies highly expressed genes while in time series of temperatures it can identify abnormal periods such as a heat wave. The core algorithm will basically slide a window of a given size over the input sequence and calculate the median for the windows. Windows for which the median deviates from some expected value are considered to be significant windows.

Figure 5.1 illustrates the SigWin-detector experiment workflow. The `LocalFileReader` simply reads the data sequence from file and outputs the file as message data. `ColumnReader` reads the input sequence by selecting a column from the multi-column data input. The `Rank` module, ranks the input and outputs the rank structure on port 1, the sorted rank vector on port 2, and the sorted rank vector without duplicates on port 3. `SWMedian` computes the moving medians of specified window sizes and outputs parameters to the slide window structure on port 1 and the computed medians on port 2. `SWMedianProb` computes a theoretical probability density for sliding window median data. `Sample2Freq` generates frequency count from sliding window data. `FDRThreshold` computes the high or low thresholds for each window size by applying false discovery rate (FDR)

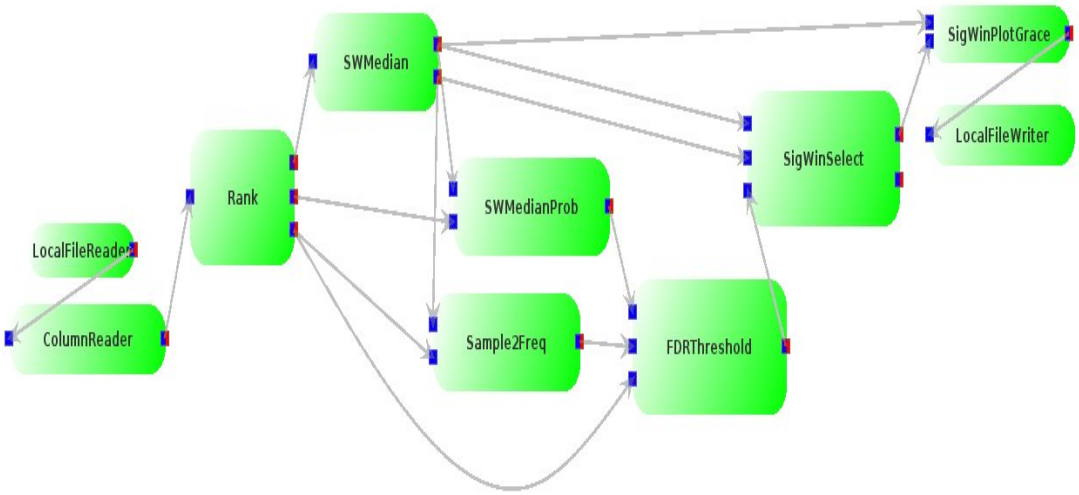


Figure 5.1: SigWin-detector base workflow

procedure that compares the frequency count from `Sample2Freq` with the theoretical probability from `SWMedianProb`. `SigWindow` selects significant windows which are above the FDR threshold. `SigWinPlotGrace` generates a plot file while the `LocalFileWriter` writes the output to local file.

The SigWin-detector was implemented from the ground up for the current WS-VLAM architecture. Communication in the current WS-VLAM modules is based on streams and messages as with our system hence the SigWin-detector modules had to be ported to match the template described in section 4.5. Much effort was taken so as not to change the scietific logic whilst porting the application. Since the current communication model is based on network streams we converted the network streams to in-memory streams and hence the modules would write to the streams as if they were network streams. The modules were then made to conform to the template by extracting reference to the data from the memory stream and generate messages. The nature of the modules is so that modules only start working once all the data has been received as in a file based approach which does not expose any computation overlap that would have been achieved from a pipeline model. Although the application is a good example for a dataflow approach since individual modules can take a considerable amount of time, it does not exploit other features available in the Datafluo systems such as farming and pipelining through messaging.

5.1.1 Results

The nature of the SigWin-detector is so that modules only produce output once when all computation has finished hence the workflow cannot exploit concurrency through computation overlap since the end of one module signals the start of the next modules. The concurrency achieved is through pure data flow where `SWMedianProb` and `Sample2Freq` modules are able to run concurrently. The modules typically consume one message per port and produce one message per output port hence farming based on messages is also restricted. For our simulation we used sample temperature data provided by the application itself. Since the workflow consists of only 10 tasks with no farming capabilities and infrequent message communication we chose to demonstrate the execution using just the bucket scheduler. We tested the workflow over 15 separate runs. Due to the relative small workflow, the resource load has no apparent impact on the workflow enactment. For this reason we do not display the resource load graphs. All workflows were clustered onto one cluster site hence the cluster distribution graph is also omitted.

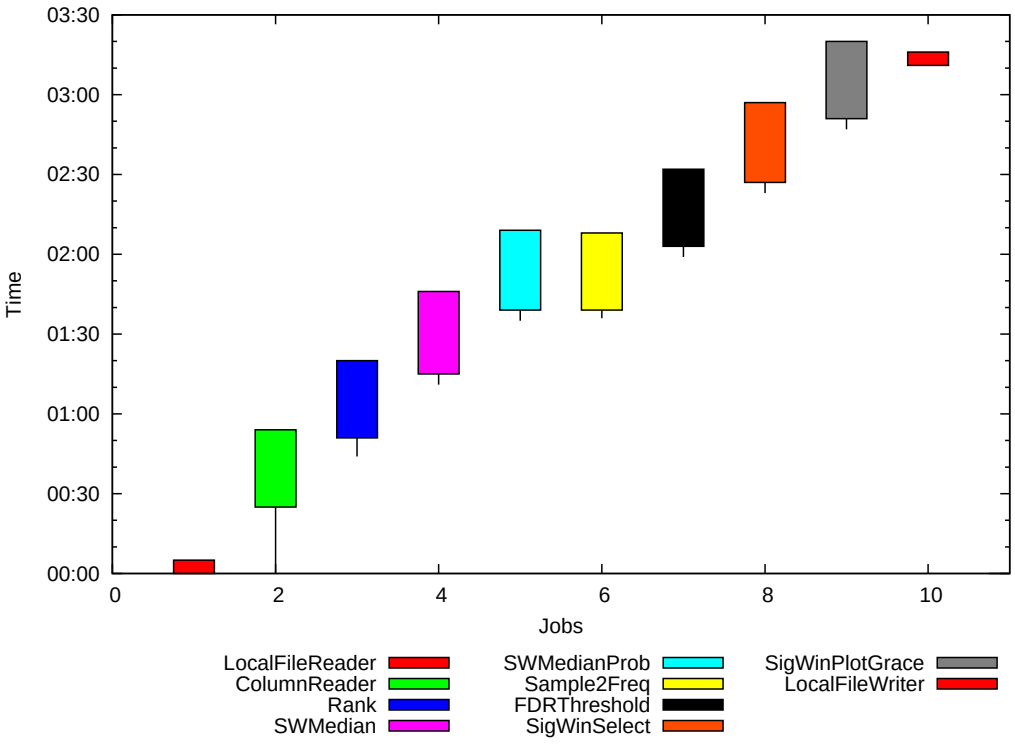


Figure 5.2: Sample run for SigWin-detector workflow

The results in figure 5.1.1 show the actual workflow enactment. As can be seen through the illustration, the modules are enacted modelling the dataflow approach where modules are scheduled for execution when data is readily available. The start module, `LocalFileReader` triggers the execution of the whole workflow. Due

to the nature of the application there is no actual computational overlap. This gives rise to the stair execution profile where tasks commence when others terminate. The slight overlap depicted between the different modules is the time a module takes to terminate after producing a result on the output port as opposed to actual computation overlap. Since the workflow only takes up a relative small resource pool, waiting times for most modules is negligible. The figure illustrates the concurrency captured through the workflow whereby both `Sample2Freq` (yellow) and `SWMedianProb` (light blue) execute concurrently. The average execution time for all the 15 samples is 204 seconds with a standard deviation of 29.1.

5.2 Wave

The Wave application deals with modelling blood flow in large vessels [10]. The application is a classic parameter study using a legacy application. The simple workflow is split into three modules as shown in figure 5.3. The `WaveParameters` act as the parameter engine for the application as it generates the parameter messages needed by the `Wave64` module. The latter module wraps around the `Wave` legacy application by gathering and creating the environment for wave to execute. The wrapper collects the legacy wave output and sends them in the form of messages to the `WaveCollector` which stores the output at a predefined location.

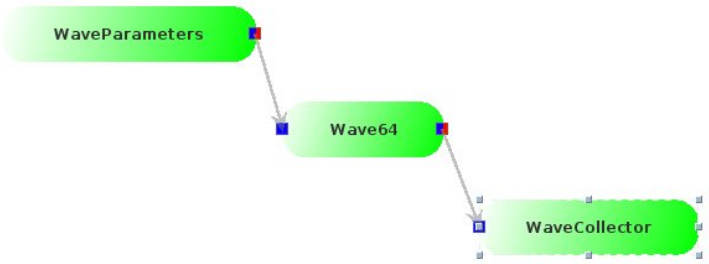


Figure 5.3: Wave

The `Wave64` is set to fixed-farming where each instance is capable of acting as a service by successively consuming parameters until, either no more parameters are available or the task has exceeded its scheduled time slot on the resource. Thus the module is said to be *greedy* as it hangs on an acquired resource as long as possible.

5.2.1 Results

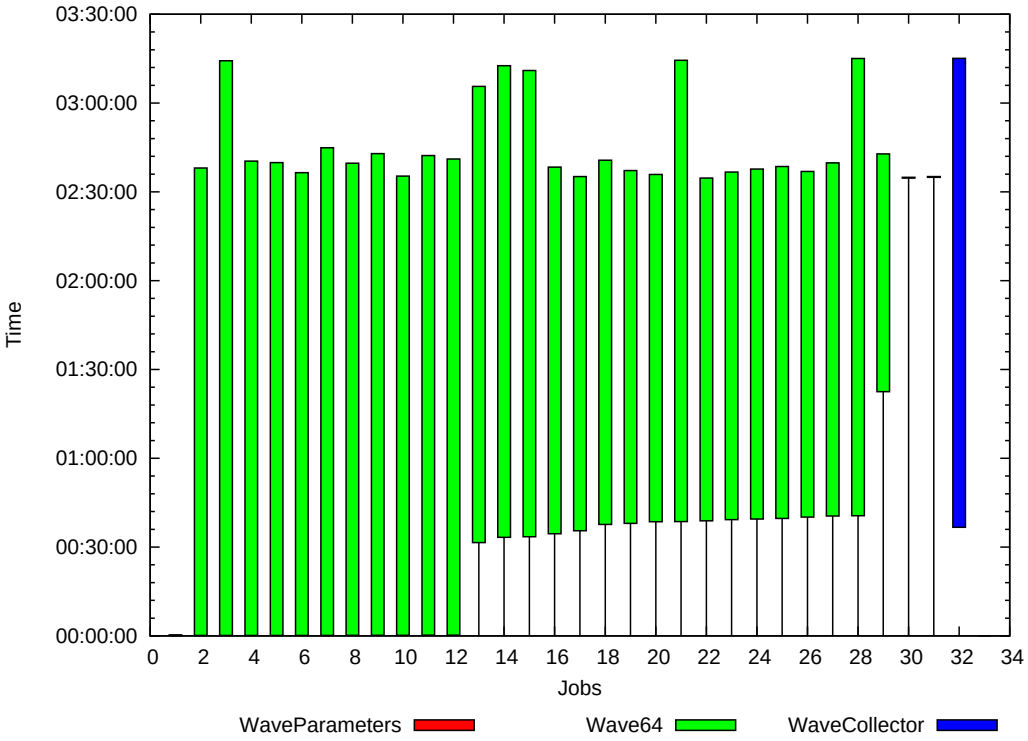


Figure 5.4: Sample Wave with farming

Figure 5.2.1 depicts the simple long running workflow enactment using fixed farming strategy. The workflow is a parameter study which covers a range of 100 parameters. The `Wave64` farm was pre-set by the user to 30 and cluster was also set to UvA so the workflow ran only on one cluster. Since the cluster at that time had only 28 working nodes, 2 `Wave64` had to wait on the queue until some resource was freed. Due to the greedy nature of the application, a resource was only freed once all parameters were exhausted so when the queue `Wave64` tasks finally executed, they had nothing to do and terminated immediately. The waiting time for tasks 13 to 29 correspond to the limited resources due to other applications running on the cluster at that time.

5.3 HistogramDifference

The `HistogramDifference` workflow application is a synthetic application built solely for testing the features implemented by the Datafluo architecture namely farming and parameter sweeping. The aim of the application is to calculate the euclidean distance between generated histograms of different colour spaces for the same image. An image histogram partitions the image colour space into bins

and groups all the pixels into one of the bins depending on the colour value hence the histogram shows the colour distribution of an image. Figure 5.5 shows the workflow layout. `DirectoryReader` recursively reads a directory and creates messages for each image file which is passed on to the `RGB2rgb` module. This module converts the raw RGB channels to a normalised form which eliminates intensity information from RGB and makes the image invariant to illumination intensity, shadows and shading. `rgb2c1c2c3` and `rgb2I1I2I3` are yet another two colour space converters. The outputs from these colour space converters is taken as input to the `HistogramDifference` which, coupled with the `Parameters`, calculates the euclidean distance between the histograms of both colour spaces. The `Parameters` module acts as the parameter sweep engine where the output is the histogram's bin size. Thus the `HistogramDifference` can calculate the euclidean distance for different histogram bin settings. The `ImageCollectors` act as intermediate data captures where the output from the image converters is kept for further analyses. Finally the `Results` module captured the workflow results.

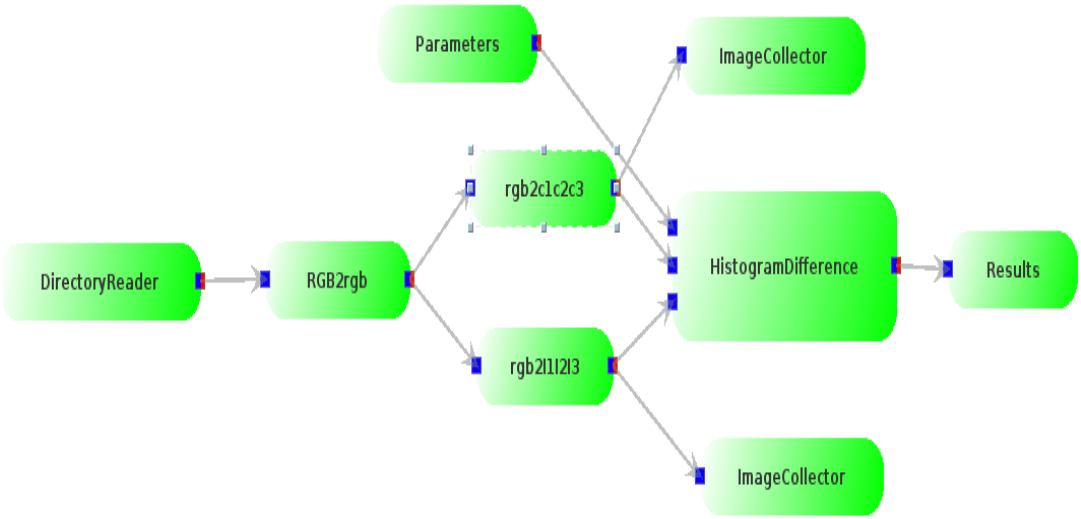


Figure 5.5: Histogram Difference

The implementation of the core modules, that is `RGB2rgb`, `rgb2c1c2c3`, `rgb2I1I2I3`, and `HitogramDiffernece` are examples of including third party software as part of a workflow system. The modules make use of GNU Octave numerical computation tool to manipulate images in through matrix transformations. The actual logic is implemented in a number of Octave functions. The template described in section 4.5 is used to wrap around the GNU Octave where the file names extracted from the messages are passed as parameters to the implemented

Octave functions. Although each Octave function has its unique wrapper, one can conceive a generic Octave wrapper which can be used for most functions. The application is intended to expose most Datafluo features within one workflow. The `rgb2c1c2c3` and `rgb2I1I2I3` are set to auto-farming hence they are able to clone themselves depending on the number of input images and the time it takes to process each image. So as to make the experiment more realistic, the `rgb2c1c2c3` modules has an induced overhead which increases the granularity of the process. The `ImageCollectors` are set to fixed-farming where each instance can clone itself once hence resulting in four `ImageCollectors`. `HistogramDifference` is set to one2one-farming where the farmed port is set to the `Paramters` port hence for every message received from the `Parameters` module a new instance of `HistogramDifference` is created. Module computation is overlapped through message pipelining. Once an image has been processed the result is sent immediately.

5.3.1 Results

The nature of the application is one that allows for computation overlap through message pipelining and different types of task farming. The communication patterns within this workflow is relatively high which is also a good test for the whole system. The workflow input dataset is a list of 390 image files obtained from the *Amsterdam Library of Object Images (ALOI)* and three parameters. The message pattern is so that the workflow generates 5463 messages from which 4920 are actual images and the remainder 1173 are text messages containing parameters and results. The workflow was tested under the different conditions: no farming, farming enabled with RoundRobin scheduler, and farming enabled with Bucket scheduler. The farming scenarios where each executed 15 times to get a good sample of the execution patterns while the non-farmed scenario was run 3 times due to its time consuming execution and little variance in the execution pattern. The mean runtime for the non-farmed scenario is 54 minutes. The execution pattern can be illustrated in figure 5.3.1. Although the modules `rgb2c1c2c3`, `ImageCollector`, `HistogramDifference` and `Results` all take over 50 minutes of processing time, the actual bottleneck is `rgb2c1c2c3` which takes long to process each image due to a purposely induced 5 second delay per image hence with 390 images the delay by itself already consumes 32.5 minutes. The other modules are the dependant modules which, after receiving the first message, have to wait for `rgb2c1c2c3` to produce messages. This results in a bottleneck.

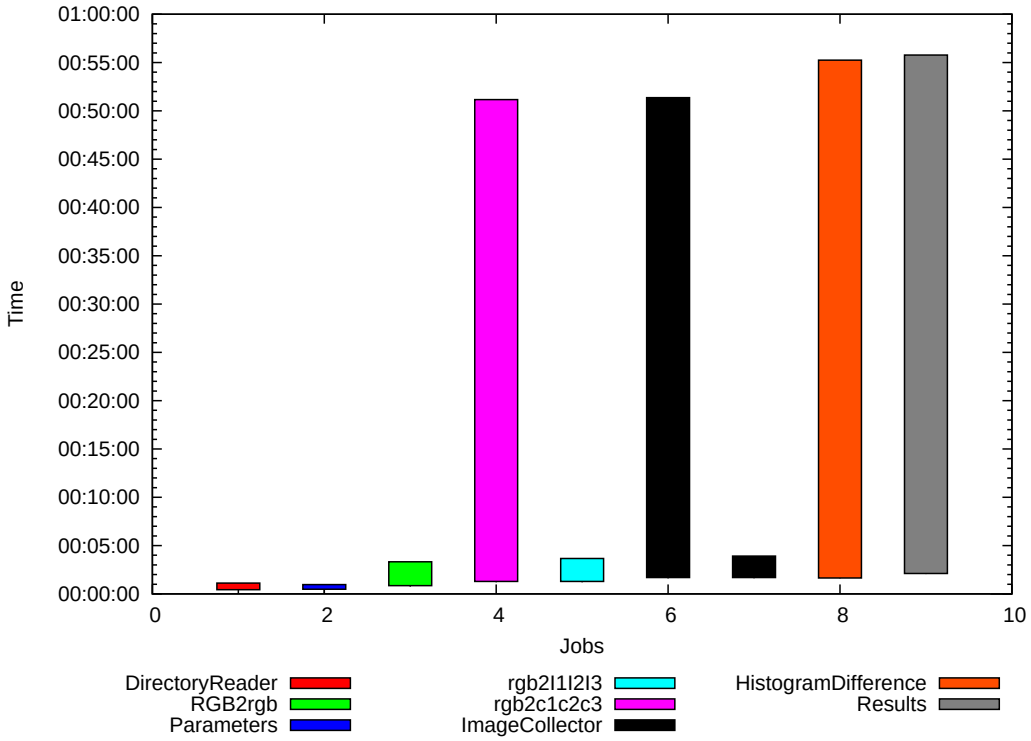
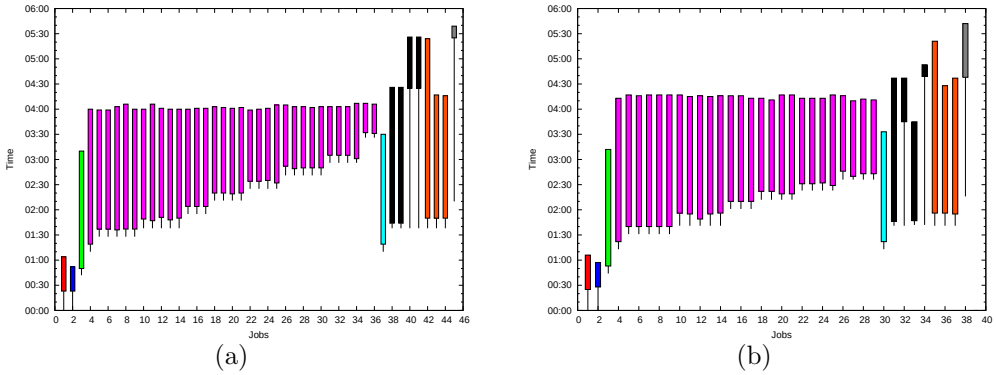


Figure 5.6: Sample HistogramDifference without farming

One way to tackle such workflow bottlenecks is through task farming where the load can be distributed amongst a farm of replicated task. These farming capabilities are shown in the following scenarios where the workflow execution time was reduced almost 10 fold. The farming scenario is further tested under the different scheduling strategies mentioned above. The mean runtime for the Bucket scheduler was 342 seconds with a standard deviation of 22 seconds while for the RoundRobin the mean time was 394 seconds with a standard deviation of 19 seconds. The Bucket scheduler proves to be slightly faster than the RoundRobin as a result of localised communication since most tasks are grouped locally hence less images are sent between cluster sites.

Figures 5.8(a) to 5.8(c) illustrate the enactment of the HistogramDifference workflow using the Bucket scheduler. Figure 5.8(a) illustrates the module enactment where each colour represents a workflow module. The latter exhibits the farming capabilities where the bottleneck task `rgb2c1c2c3` is now replicated 34 times hence better dealing with the access load. The farm profile is so because the auto-farming incrementally issues new clones hence successive clones have less work to do. The `rgb2I1I2I3` module was also set to auto-farming but since it was fast enough to consume the messages no farming was needed. The `ImageCollector` is duplicated once for each instance hence having 4 in total. The `ImageCollector` is an example of fixed farming where the user indicates

how many clones should be initiated. The `HistogramDifference` task is set to one2one-farming hence it is replicated for each parameter message received which happen to be 3. All in all, the different farming techniques expanded the original 9 task workflow into 46 task workflow. Figure 5.8(b) illustrates the task distribution on the different clusters. Since the scheduler was set to the Bucket, all tasks got scheduled on one resource at TU Delft. The tasks scheduled on the UvA cluster are intentionally bound by the user since the input data resides on the cluster while the output location is also on the specified cluster. Figure 5.8(c) shows the resource queue loads during our execution which clearly depicts the TU Delft spike due to the Bucket scheduler.



(a) (b)

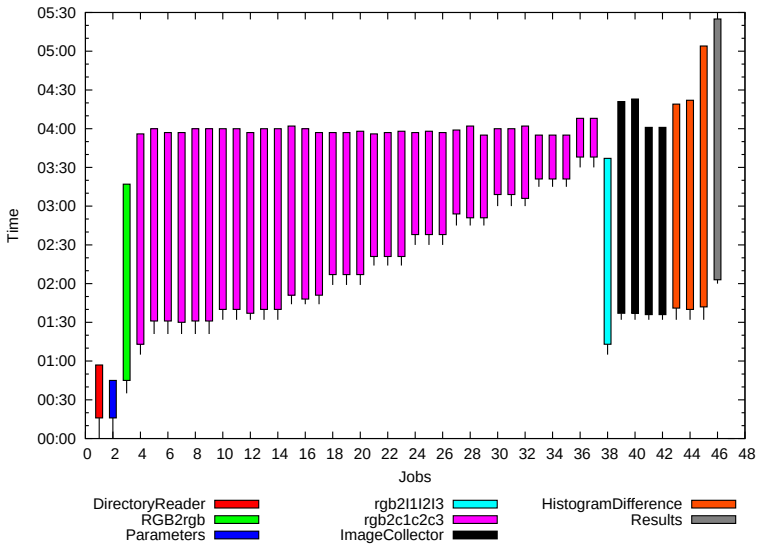
Figure 5.7: HistogramDifference Orchestration

As with the Bucket scheduler example, figures 5.9(a) to 5.9(c) show the same HistogramDifference workflow under the RoundRobin scheduler. From figure 5.9(a), the module execution pattern is not much different than that described using the Bucket scheduler.

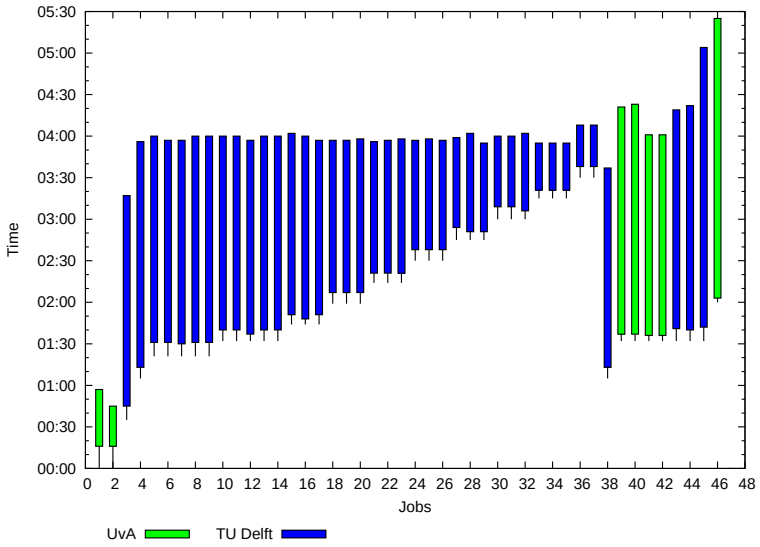
In this particular execution the `rgb2c1c2c3` farm consisted of 28 clones. One can also notice that `rgb2I1I2I3` task was also cloned. Looking at the resource distribution on figure 5.9(b) we notice that the first instance of `rgb2I1I2I3` was scheduled on the Liacs cluster. A possible explanation for this sporadic cloning could be that the communication overhead between cluster sites at that particular time triggered the extra clones to be scheduled. In this particular scenario the calculated task submission ratio for the RoundRobin scheduler was 1:1:1:2:1 for VU, Liacs, UvA, TU Delft, and Mult-UvA respectively hence each resource gets 1 task on every round of submissions except TU Delft which gets 2 tasks on every round. This strategy is illustrated in figure 5.9(b) where TU Delft (in blue) gets two tasks at the same time. The UvA resource is a special case where some tasks are purposely bound to the resource. Figure 5.9(c) show the load on all clusters during execution. From the figure we can notice that most clusters

already had some load before our execution such as VU which was at 50% load at start time. One can notice that the load imposed by our scheduler is spread out across the different resources. The access load on the UvA resource is due to the `DirectoryReader`, `Parameters`, `ImageCollector` and `Results` task binding.

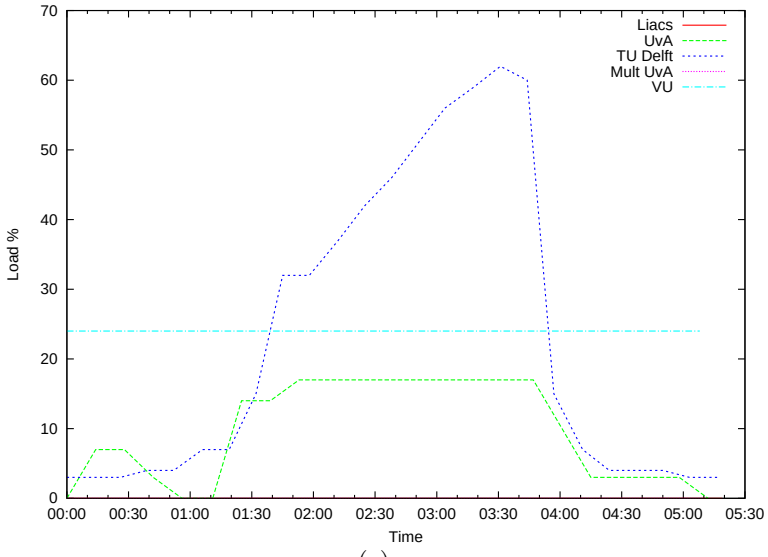
Figures 5.7(a) and 5.7(b) depict the effect of resource exhaustion on the workflow enactment. We can notice the in both figures several jobs (`ImageCollector` and `Results`) are stuck waiting. This is due to not having enough computing nodes to schedule the workflow. In figure 5.7(a) we can notice that the `Results` module takes a relatively small execution time. The reason for this is that by the time the module gets to start execution all the result messages are ready waiting on the queue hence the module has no idling time and can consume all the results messages immediately. The same effect can be noticed with the `ImageCollector` module in figure 5.7(b). In addition since the `ImageCollector` is farmed and one of the farmed instances got scheduled on the resource, the running instance offloaded the waiting instance hence once the waiting module got scheduled it had much less work to do.



(a)

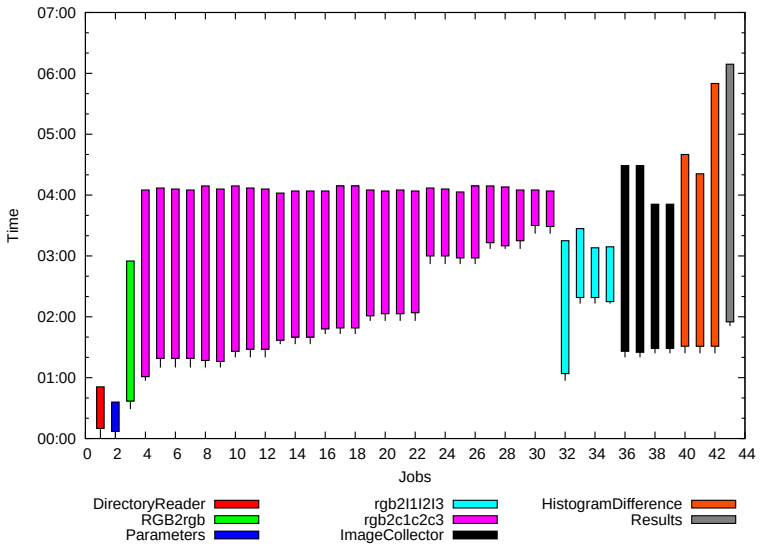


(b)

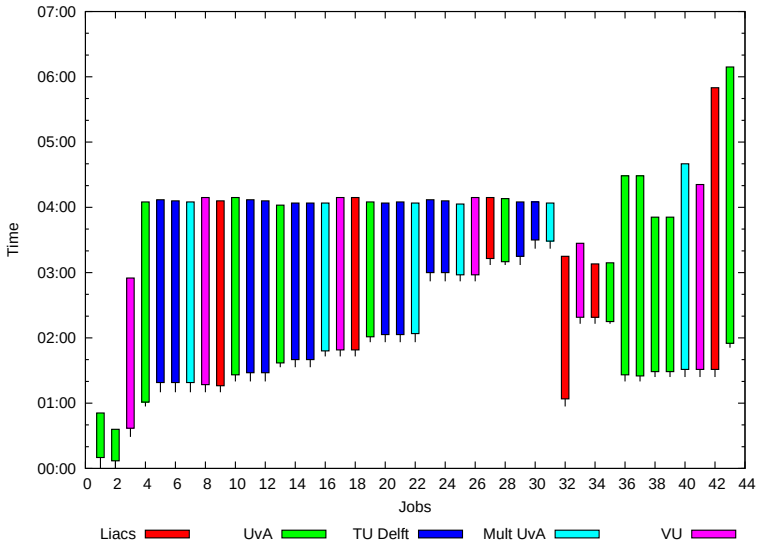


(c)

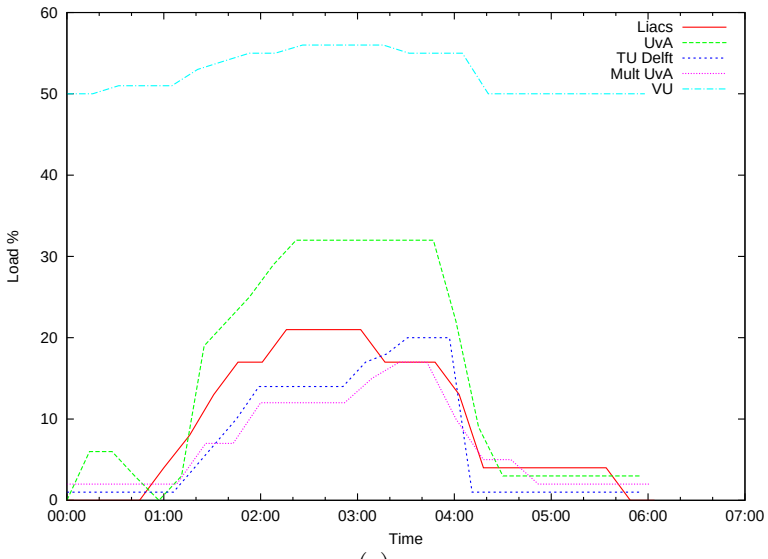
Figure 5.8: Sample HistogramDifference with Bucket scheduler.



(a)



(b)



(c)

Figure 5.9: Sample HistogramDifference with RoundRobin scheduler.

Chapter 6

Conclusions and Future Work

6.1 Future Work

As for the architecture, Datafluo still lacks some important features such as a concrete fault tolerance system and provenance system. The current fault tolerance system is based on task resubmission in the event that the task terminates unexpectedly. The architecture does not capture state hence once a task is resubmitted all state is lost. One idea of integrating state information into the system is to implement special input and output state ports as depicted at the top layer in figure 6.1. The output state port can be used by the task to send state information which can be considered as checkpoint information. Every time a task is initiated its state port is checked in which case state information is retrieved and the task can continue from where it ended. Having such a feature will not only facilitate fault tolerance but also opens the door to implement new features into the architecture such as a migration system where Datafluo can pre-empt tasks and decide to migrate them to other resources. This could be beneficial in situations where tasks can be migrated closer to the data or cluster tasks together at runtime.

Typical middleware scheduling systems allocate a time quantum for each task with the aim of giving a fair opportunity to all tasks on the queue. This time quantum poses a challenge since tasks that have exhausted their allocated time slot would pre-emptively be terminated by the scheduler. To better handle the allocated time, a task can be made self aware of its allocated time and before starting to process a message decides if it has enough time to do so or not in which case it will gracefully exit and signal the Datafluo server to re-submit the task. The assumption in this scenario is that tasks take the same approximate time to process each message hence they can predict if there is enough time to continue processing new messages.

Datafluo architecture could also benefit from control flow. As it is difficult to express iteration with a dataflow model, control flow can be superimposed onto the data flow layer. In this scenario, the control blocks represent dataflow graphs and

the control system merely coordinates the sub dataflow blocks such as allowing dataflow iterations. This is illustrated at the middle layer in figure 6.1.

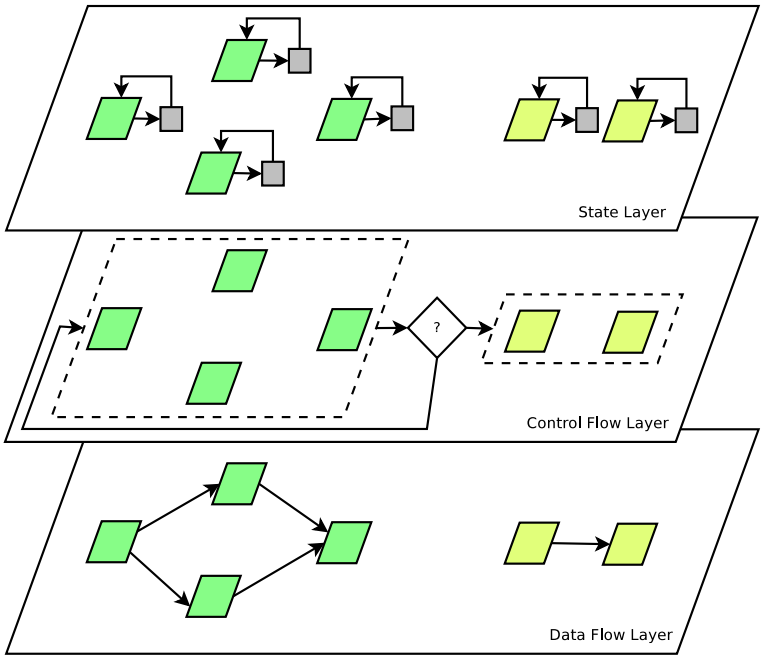


Figure 6.1: The bottom layer represent the current data flow implementation. The second layer represents a control flow layer where segregated workflows are joined together through control structures. The top layer represents module state management at a workflow level where each module can write its state to the WFMS and thus allow for better fault tolerance and scheduling through migration.

The Datafluo architecture also lacks hierarchical workflow composition which facilitates complex workflow composition whilst also aiding in a distributed coordinating system where each node in the hierarchy could be coordinated by a separate Datafluo instances. In this case the message exchange would act as the message router between different Datafluo instances hence allowing tasks to communicate on a distributed coordinated system. Since message communication can overwhelm the Datafluo server system, the architecture could benefit from a system where messages from tasks on a local cluster are aggregated and sent to the server as opposed to the situation where every task communicates directly to the server. In this scenario, a group of tasks on a local cluster elect a task that coordinates all communication with the server.

As it is, the message exchange system keeps messages in memory which is not ideal for long running workflows that produce many messages hence a message back-end would be beneficial where messages can be stored in a persistent storage.

A provenance system can be also attached to the message exchange since messages represent the state of the workflow at a particular time.

The architecture is intended to employ many forms of distributed resources, thus not restricting the WFMS to work solely on DAS3. As long as a submitter can be written for some resource, Datafluo can be made to *mix-and-match* modules to different resources. Ongoing work is currently being done to use Cloud resources alongside the DAS3 clusters also, GMinion, a job management system for production grids is currently being integrated as another resource. Another possibility is to integrate volunteer computing such as BIONIC into Datafluo. With volunteer computing, one can conceive of a scheduler with prioritise computation intensive tasks to be scheduled on the volunteer resources while frequent communicating tasks are kept close together on other resources.

6.2 Conclusions

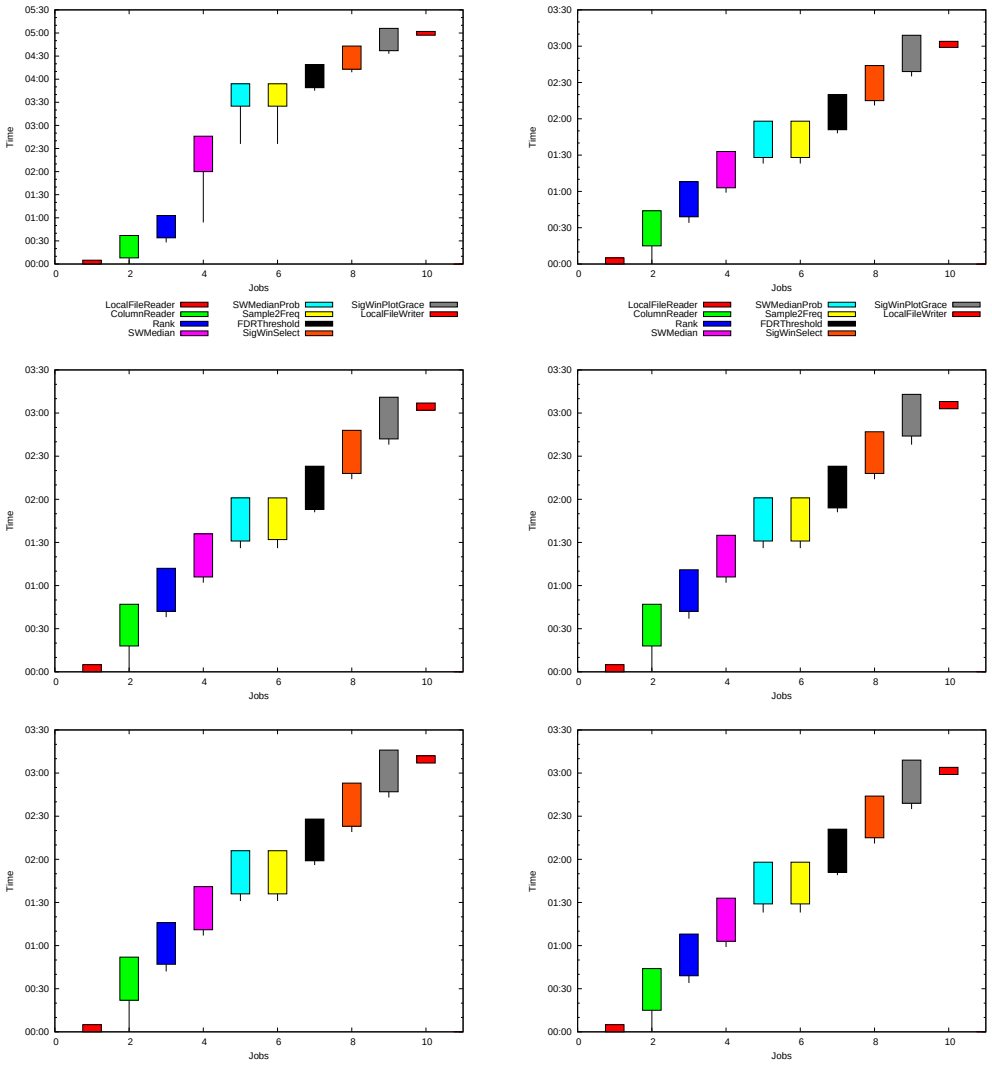
In this dissertation we showed how applying dataflow, pipeline and farming concepts to a scientific workflow system on grid architectures can be advantageous for common scientific problems characterised by long running embarrassingly parallel tasks. These concepts were the building blocks for our Datafluo architecture. The results show how task startup idling time is eliminated by scheduling tasks with a dataflow model. The results also show how decoupling task communication can ease the task schedule. In scenarios such as the `HistogramDifference` workflow we showed how farming concepts can drastically reduce the execution time by replicating relatively slow tasks so as to increase the message throughput. The workflow also illustrates how farming techniques are used in parameter sweeping applications where a workflow component acts as a parameter engine as opposed to having a one-size-fits-all central parameter engine.

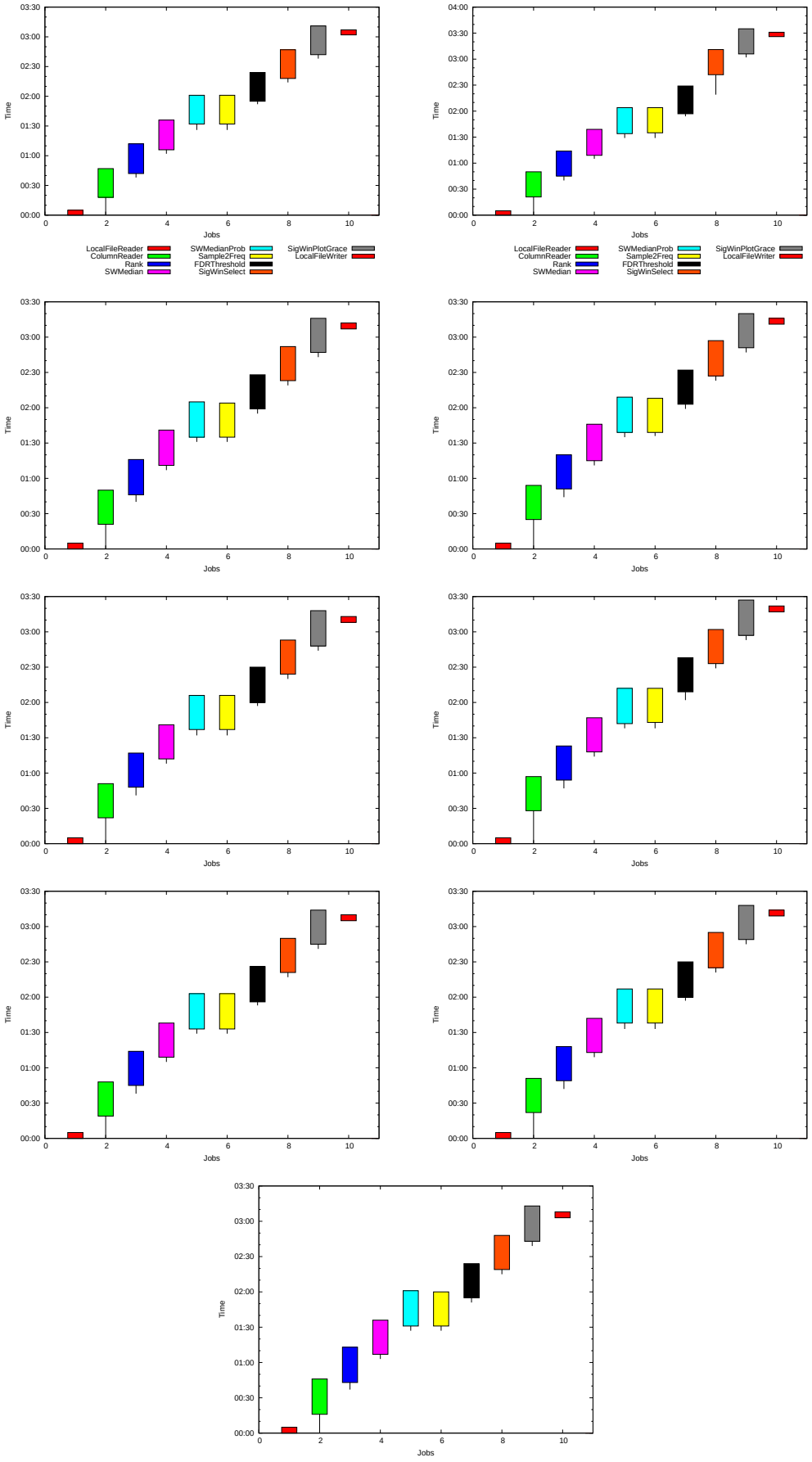
Although our Datafluo architecture implementation is a working prototype it still lacks important features such as a reliable fault tolerance system and a provenance system. Other features such as control flow and state management, we believe, will ameliorate the overall architecture.

Appendix A

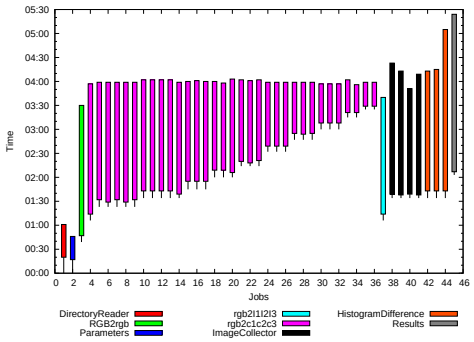
Extra Results

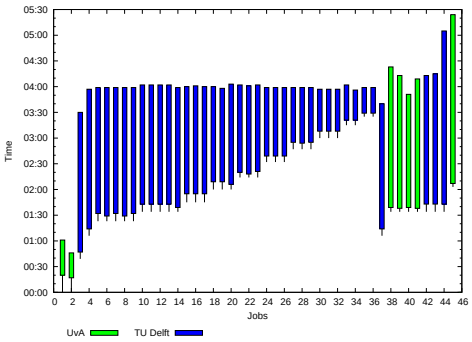
A.1 SigWin-detector

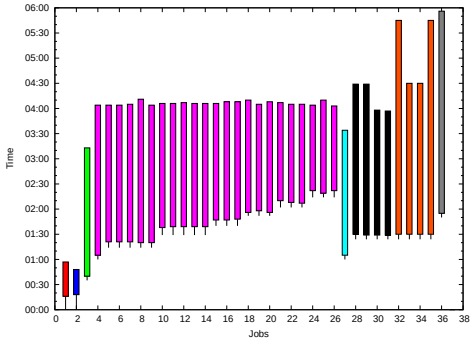


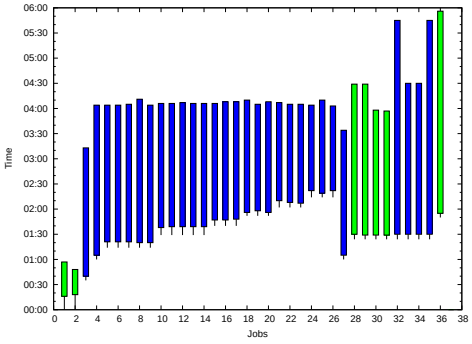

Time
Jobs
LocalFileReader
ColumnReader
Rank
SWMedian
SWMedianProb
Sample2Freq
FDRThreshold
SigWinSelect
SigWinPlotGrace
LocalFileWriter

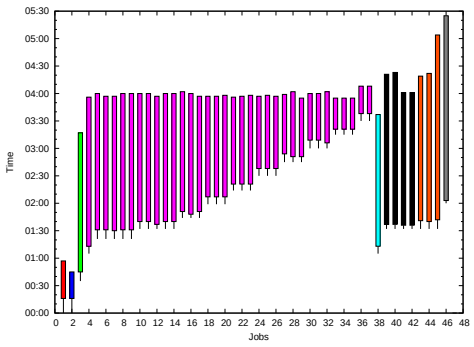
A.2 HistogramDifference - Bucket

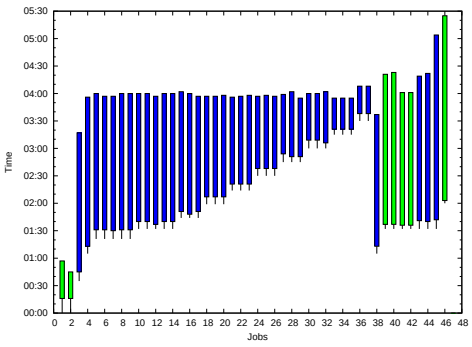


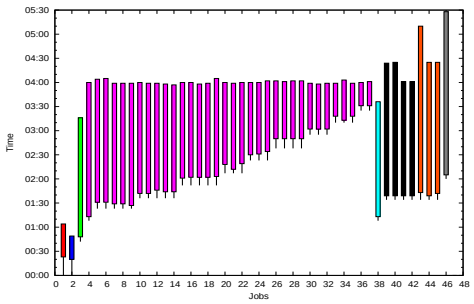


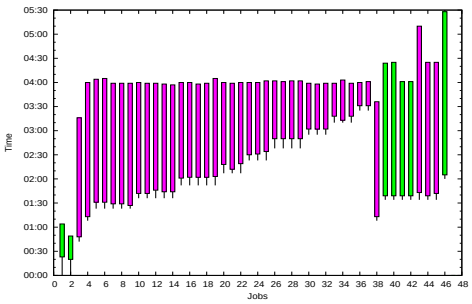


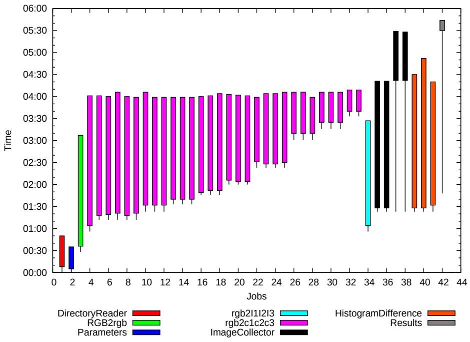


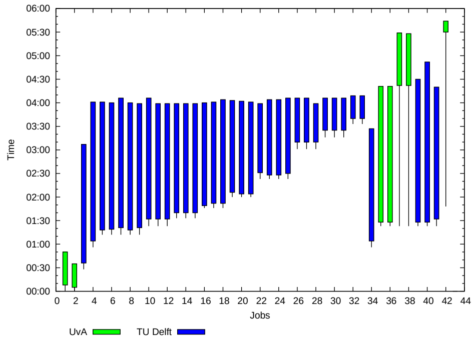


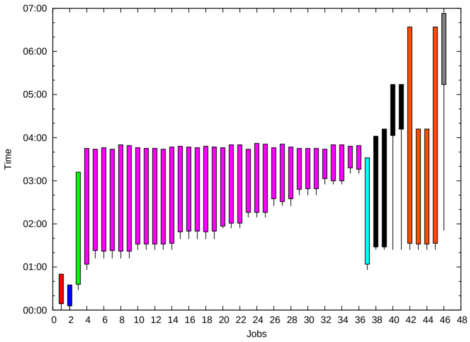


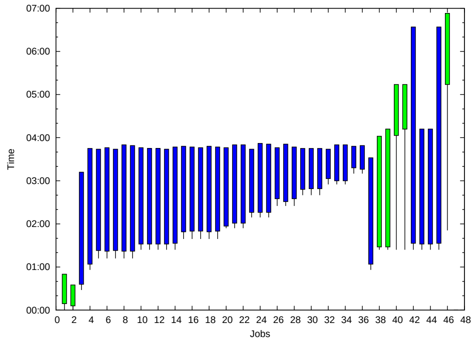


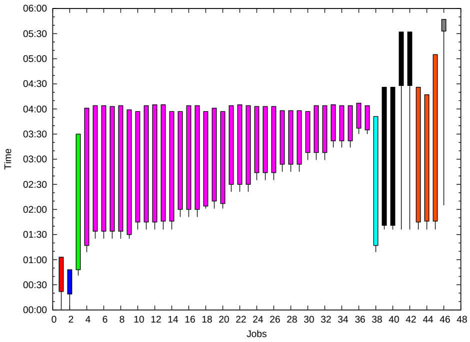


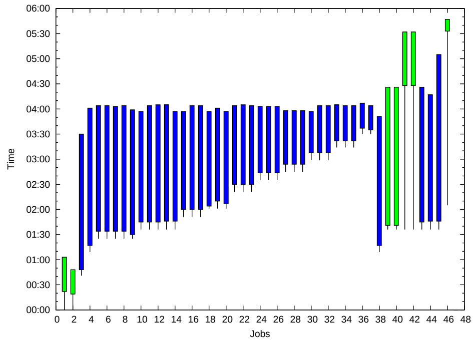

Time
Jobs
DirectoryReader
RGB2rgb
Parameters
rgb2l1l2l3
rgb2c1c2c3
ImageCollector
HistogramDifference
Results

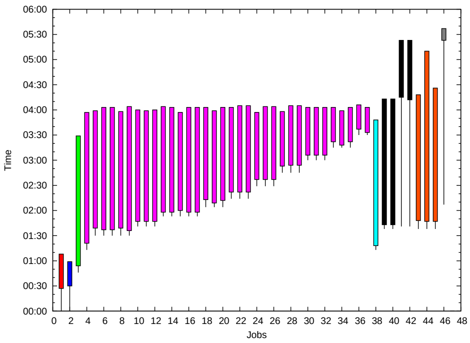

Time
Jobs
UvA
TU Delft

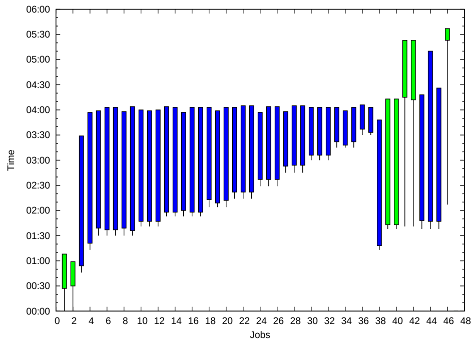

Time
Jobs

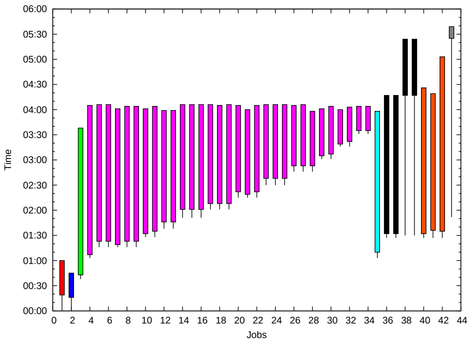

Time
Jobs

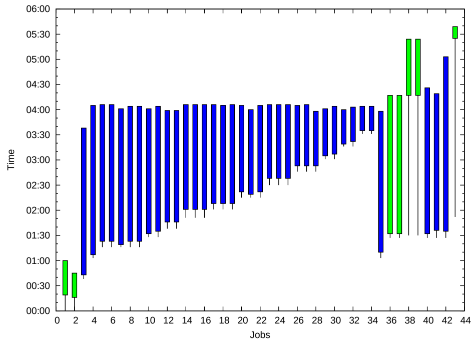

Time
Jobs

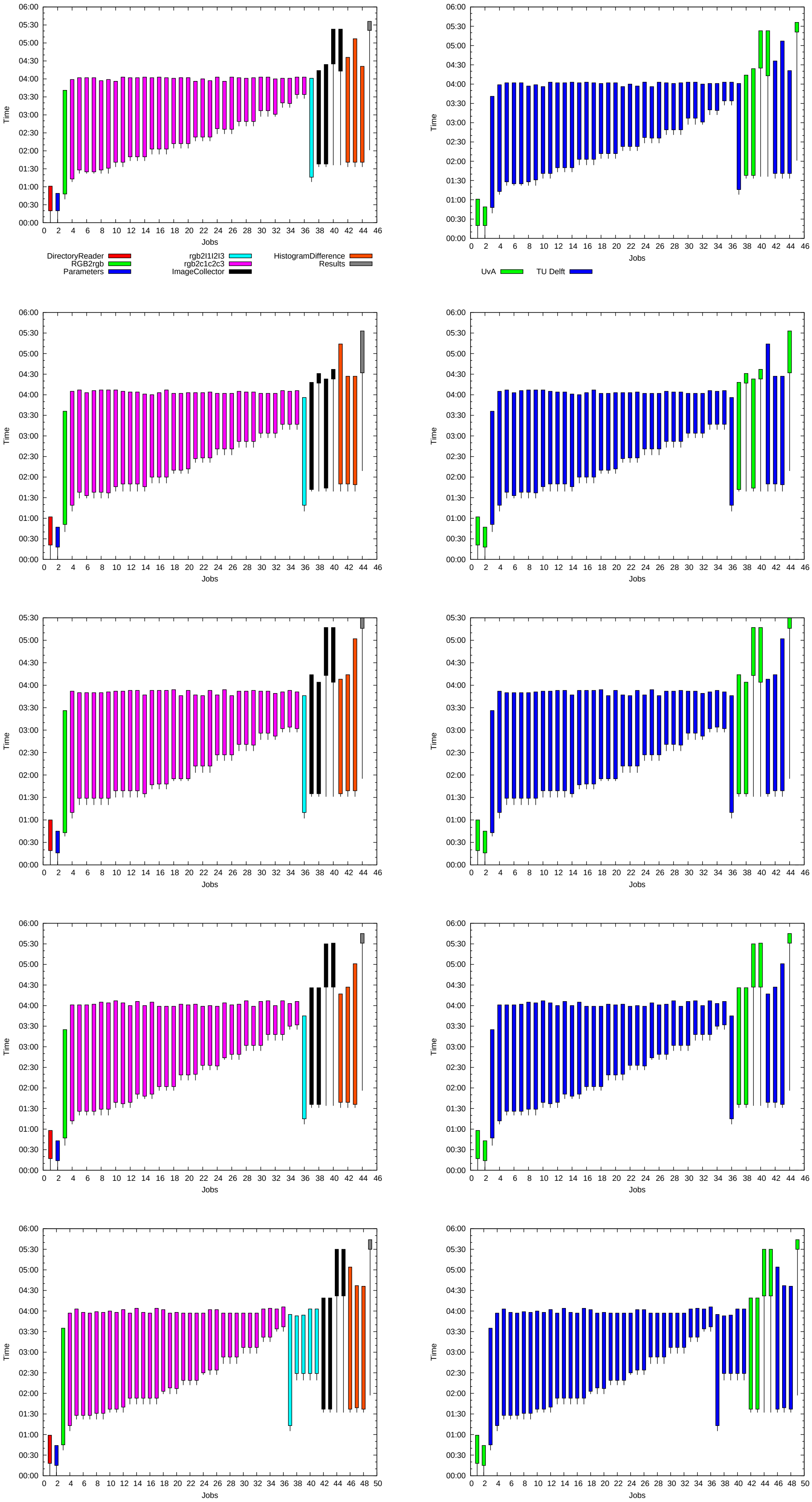

Time
Jobs

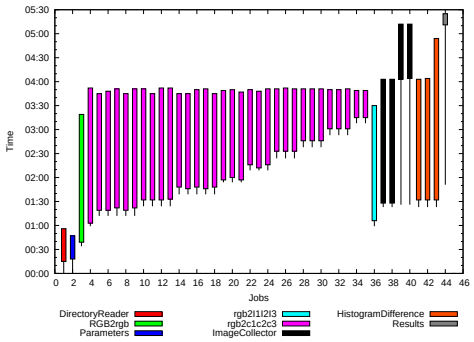

Time
Jobs

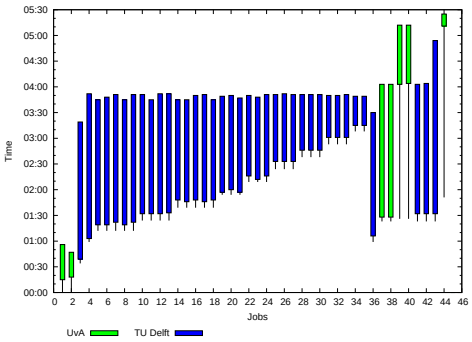

Time
Jobs


Time
Jobs

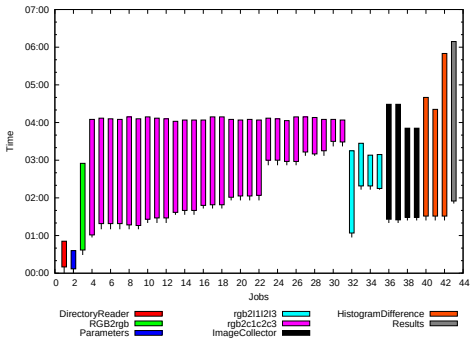

Time
Jobs

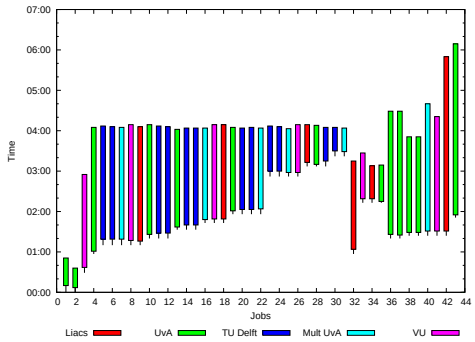

Time
Jobs
DirectoryReader
RGB2rgb
Parameters
rgb2l1l2l3
rgb2c1c2c3
ImageCollector
HistogramDifference
Results
UvA
TU Delft

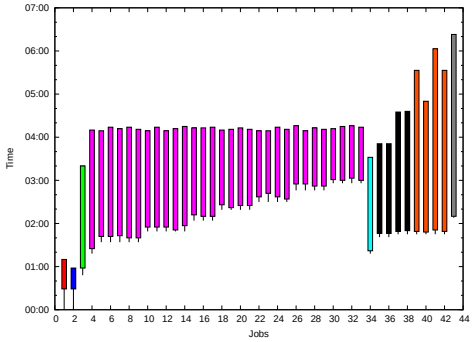

Time
05:30
05:00
04:30
04:00
03:30
03:00
02:30
02:00
01:30
01:00
00:30
00:00
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46
Jobs
DirectoryReader
RGB2rgb
Parameters
rgb2I1I2I3
rgb2c1c2c3
ImageCollector
HistogramDifference
Results

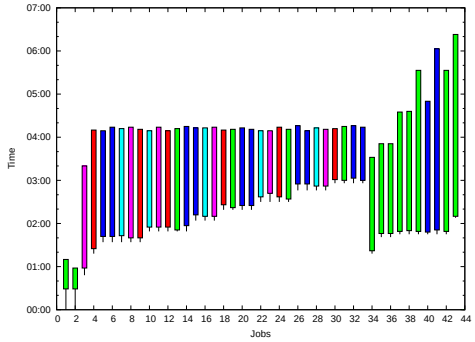

Time
05:30
05:00
04:30
04:00
03:30
03:00
02:30
02:00
01:30
01:00
00:30
00:00
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46
Jobs
UvA
TU Delft

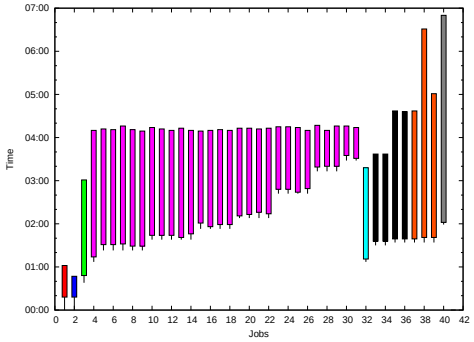
A.3 HistogramDifference - RoundRobin

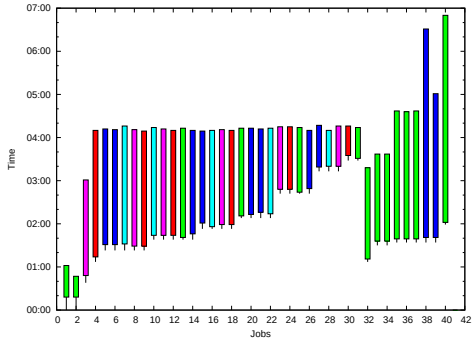


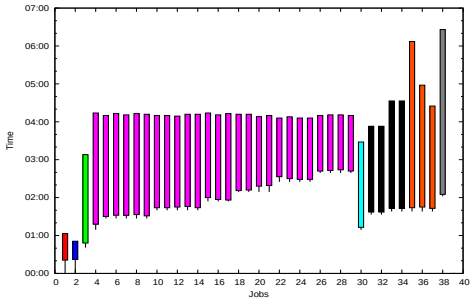


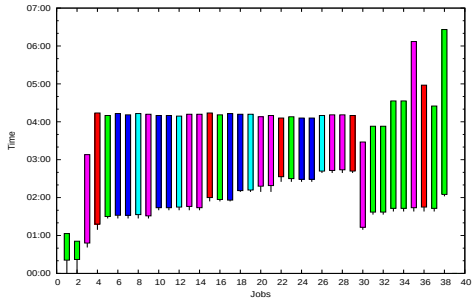


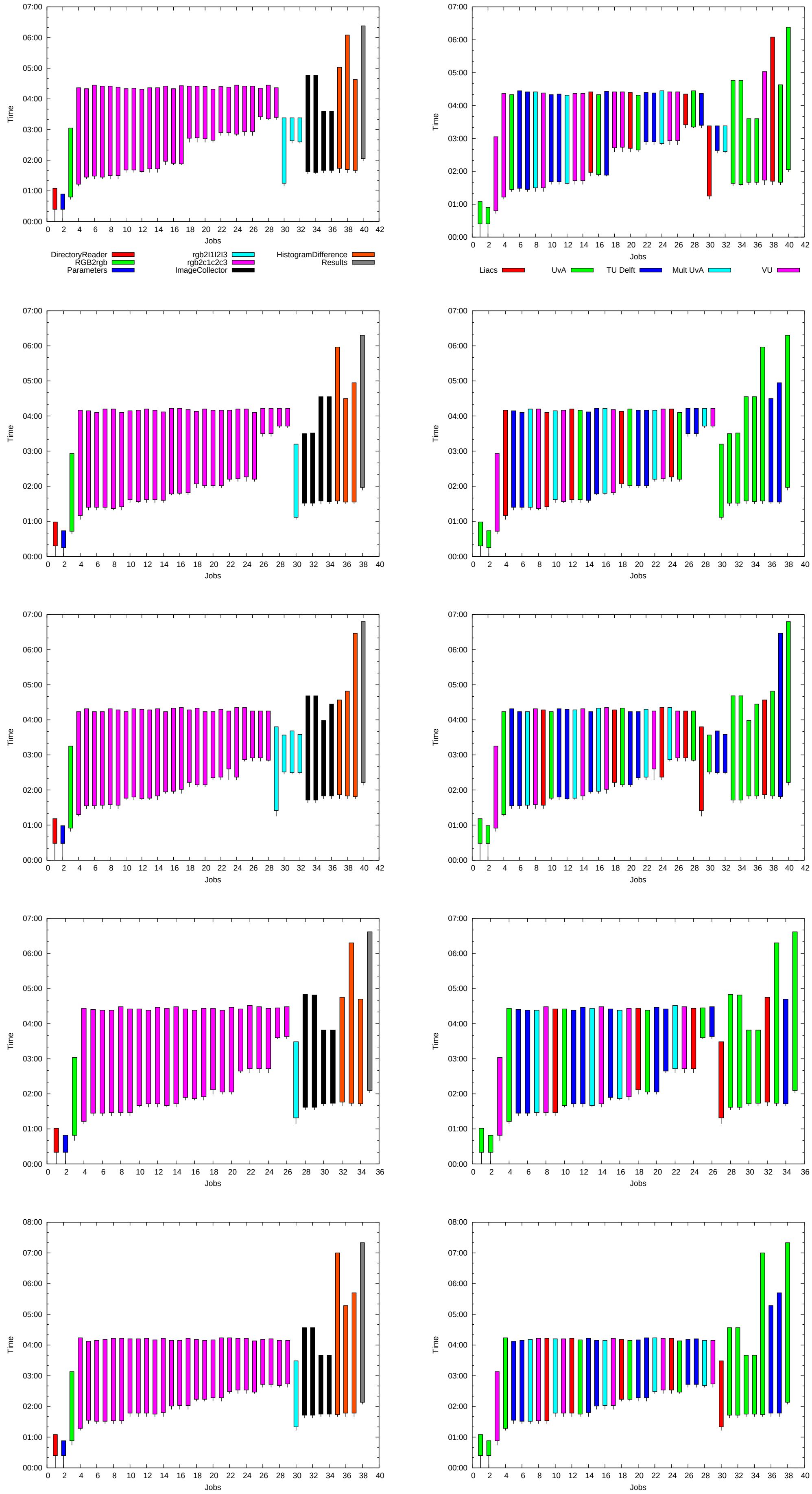


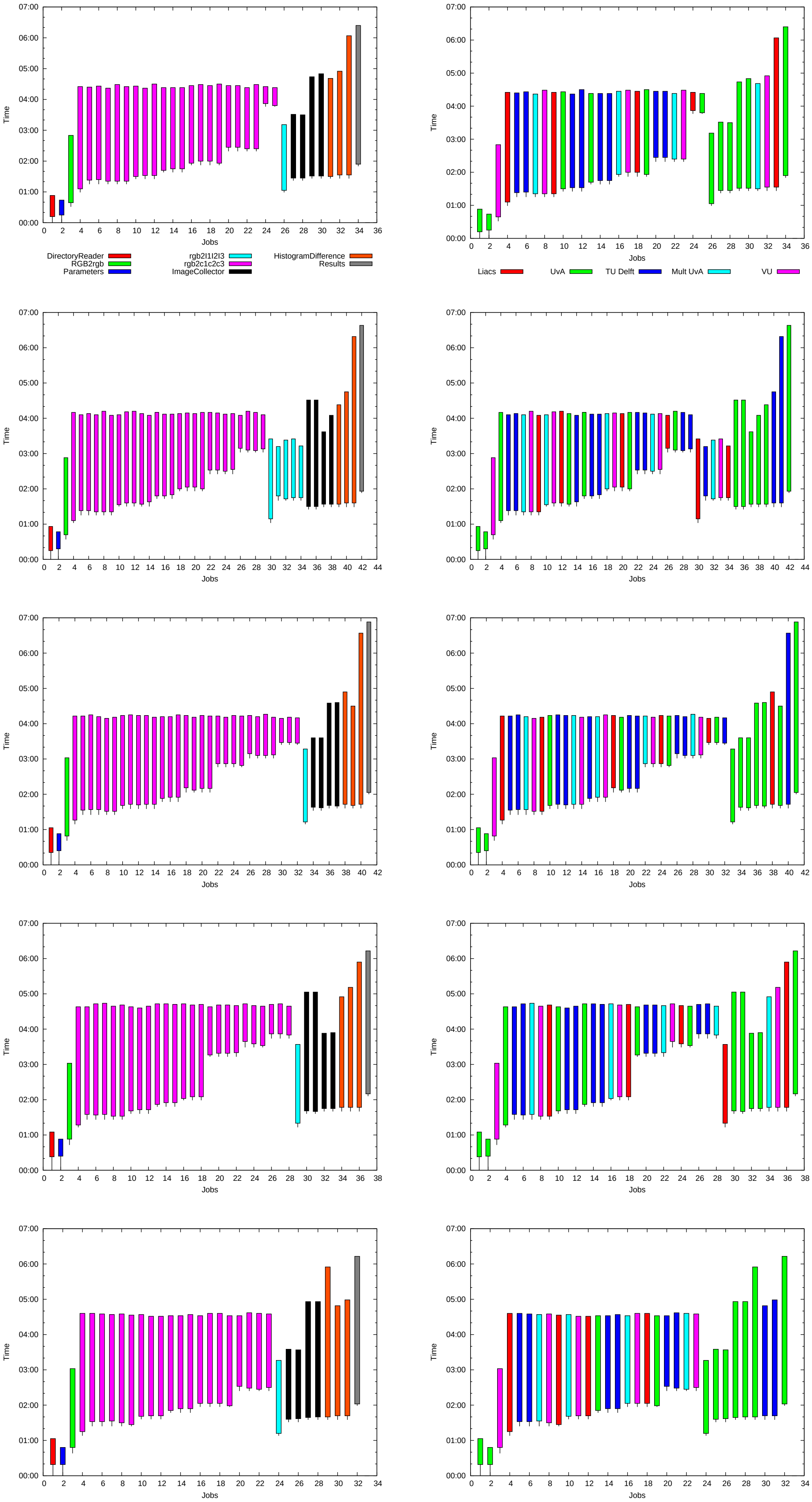


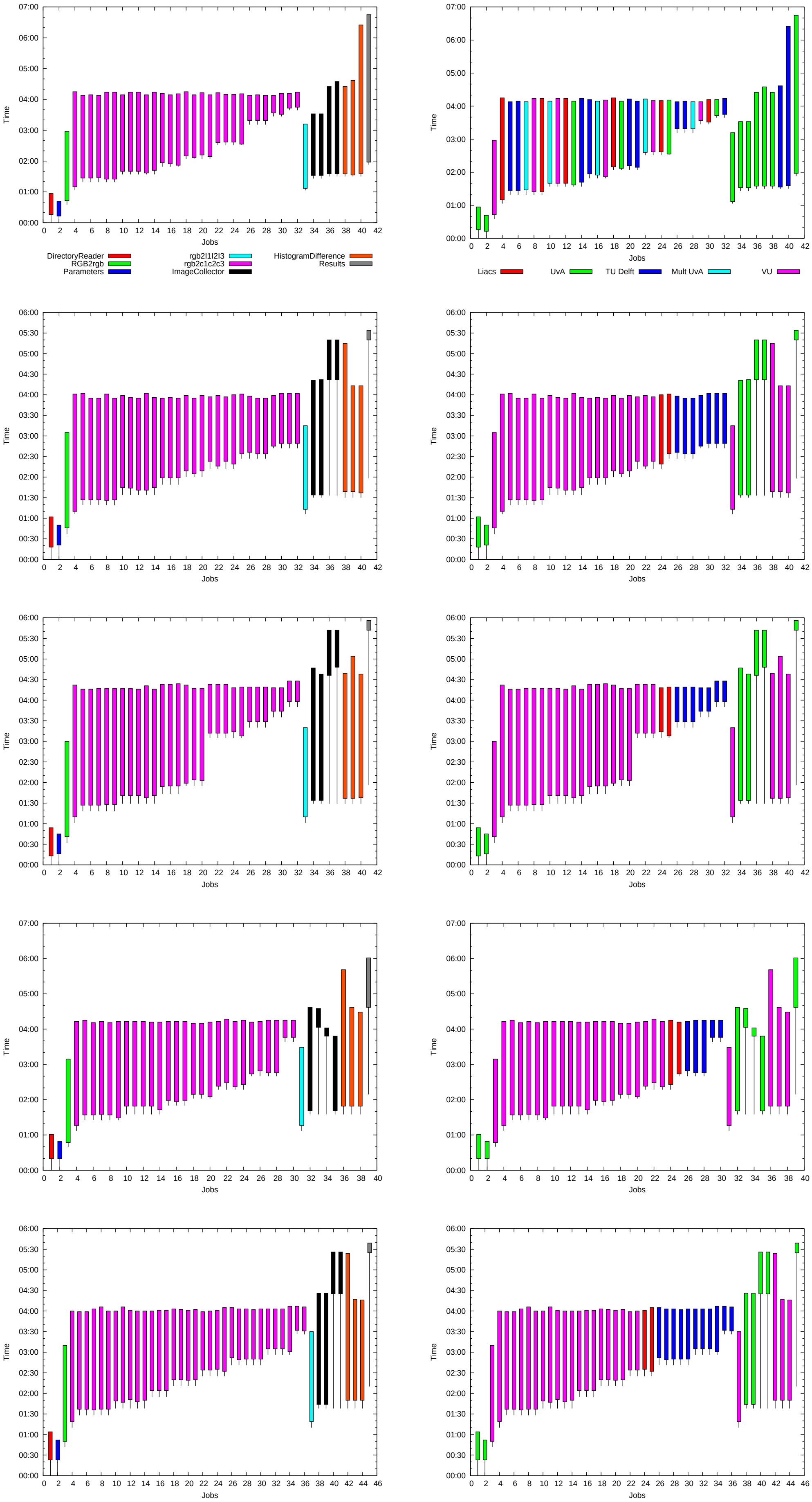







Time
Jobs
DirectoryReader
RGB2rgb
Parameters
rgb2l1l2l3
rgb2c1c2c3
ImageCollector
HistogramDifference
Results
Liacs
UvA
TU Delft
Mult UvA
VU


Time
Jobs
DirectoryReader
RGB2rgb
Parameters
rgb2l1l2l3
rgb2c1c2c3
ImageCollector
HistogramDifference
Results
Liacs
UvA
TU Delft
Mult UvA
VU


Time
Jobs
DirectoryReader
RGB2rgb
Parameters
rgb2l1l2l3
rgb2c1c2c3
ImageCollector
HistogramDifference
Results
Liacs
UvA
TU Delft
Mult UvA
VU

Bibliography

[1] Enabling Grids for E-sciencE (EGEE). http://eu-egee.com.

[2] Freefluo Workflow Enactor. http://freefluo.sourceforge.net.

[3] Maui cluster scheduler. http://www.clusterresources.com/products/maui-cluster-scheduler.php.

[4] myExperiment. http://www.myexperiment.org/.

[5] National e-Science Centre Uk. http://www.nesc.ac.uk/nesc/define.html.

[6] H. Abbasi, M. Wolf, K. Schwan, G. Eisenhauer, and A. Hilton. Xchange: coupling parallel applications in a dynamic environment. *Cluster Computing, IEEE International Conference on*, 0:471–480, 2004.

[7] David Abramson, Blair Bethwaite, Colin Enticott, Slavisa Garic, Tom Peachey, Anushka Michailova, Saleh Amirriazi, and Ramya Chitters. Robust workflows for science and engineering. In *MTAGS '09: Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, pages 1–9, New York, NY, USA, 2009. ACM.

[8] M Addis, J Ferris, M Greenwood, P Li, D Marvin, T Oinn, and A Wipat. Experiences with e-science workflow specification and enactment in bioinformatics. In S.J. Cox, editor, *e-Science All Hands Meeting 2003*, pages 459–466, 2003.

[9] Ishfaq Ahmad and Yu kwong Kwok. On exploiting task duplication in parallel program scheduling, 1998.

[10] D. Bessems, M. Rutten, and F. N. van de Vosse. A wave propagation model of blood flow in large vessels using an approximate velocity profile function. *J. Fluid Mech.*, 580:145–168, 2007.

[11] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid. pages 283–289. IEEE Computer Society Press, 2000.

[12] Manuel Caeiro-Rodriguez, Thierry Priol, and Zsolt Németh. Dynamicity in scientific workflows. Technical Report TR-0162, Institute on Grid Information, Resource and Workflow Monitoring Services , CoreGRID - Network of Excellence, August 2008.

[13] Scott Callaghan, Ewa Deelman, Dan Gunter, Gideon Juve, Philip Maechling, Christopher Brooks, Karan Vahi, Kevin Milner, Robert Graves, Edward Field, David Okaya, and Thomas Jordan. Scaling up workflow-based applications. *Journal of Computer and System Sciences*, In Press, Corrected Proof:–, 2009.

[14] Henri Casanova and Fran Berman. Concurrency: Pract. exper. 2002; parameter sweeps on the grid with apst, 2002.

[15] Henri Casanova, Graziano Obertelli, Francine Berman, and Richard Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid, 2000.

[16] David Churches, Gabor Gombas, Andrew Harrison, Jason Maassen, Craig Robinson, Matthew Shields, Ian J. Taylor, and Ian Wang. Programming scientific and distributed workflow with triana services. *Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, August 2006.

[17] Ewa Deelman, James Blythe, A Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei hui Su, Karan Vahi, and Miron Livny. Pegasus: Mapping scientific workflows onto the grid. pages 11–20, 2004.

[18] Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528 – 540, 2009.

[19] Hans-Joachim Bungartz Ekaterina Elts and Jadran Vrabec. Fast elaboration of molecular models using Grid workflows. *Joint Session of the ProcessNet Working Party Molecular Modelling and Simulation for Process and Product Design and the EFCE Working Party on Thermodynamics and Transport Properties*, 2010.

[20] Erik Elmroth, Francisco Hernandez, and Johan Tordsson. Three fundamental dimensions of scientific workflow interoperability: Model of computation, language, and execution environment. *Future Generation Computer Systems*, 26(2):245 – 256, 2010.

[21] Fabrizio Pacini et al. EGEE WMS Service. `https://edms.cern.ch/document/572489/1`, 2006.

[22] I. Altintas et al. Kepler: an extensible system for design and execution of scientific workflows. *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 423–424, 2004.

[23] Martin Adolph et al. Distributed Computing: Utilities, Grids & Clouds. Technical report, International Telecommunication Union, 2009.

[24] T. Fahringer et al. Askalon: A grid application development and computing environment. In *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 122–131, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Martin Feller, Ian Foster, and Stuart Martin. Gt4 gram: A functionality and performance study, 2007.

[26] Ian T. Foster. The anatomy of the grid: Enabling scalable virtual organizations. In *Proc. First IEEE International Symposium on Cluster Computing and the Grid (1st CCGRID'01)*, pages 6–7, Brisbane, Australia, May 2001. IEEE Computer Society (Los Alamitos, CA).

[27] Geoffrey Fox and Dennis Gannon. Workflow in grid systems, February 06 2008.

[28] Matthieu Gallet, Loris Marchal, and Frdric Vivien. Efficient scheduling of task graph collections on heterogeneous resources.

[29] Dennis Gannon, Sriram Krishnan, Liang Fang, Gopi Kandaswamy, Yogesh Simmhan, and Aleksander Slominski. On building parallel & grid applications: Component technology and distributed services. *Cluster Computing*, 8(4):271–277, 2005.

[30] Antoon Goderis, Christopher Brooks, Ilkay Altintas, Edward A. Lee, and Carole A. Goble. Heterogeneous composition of models of computation. *Future Generation Comp. Syst*, 25(5):552–560, 2009.

[31] Andrew Harrison, Ian Taylor, Ian Wang, and Matthew Shields. Ws-rf workflow in triana. *Int. J. High Perform. Comput. Appl.*, 22(3):268–283, 2008.

[32] Thomas Heinis, Cesare Pautasso, and Gustavo Alonso. Design and evaluation of an autonomic workflow engine. In *ICAC*, pages 27–38. IEEE Computer Society, 2005.

[33] Andreas Hoheisel. Grid workflow execution service - user manual, 2006.

[34] Andreas Hoheisel. User tools and languages for graph-based grid workflows: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1101–1113, 2006.

[35] D Hull and M Pocock. Taverna a tool for building and running workflows of services, January 01 2006.

[36] Marcia Inda, Marinus van Batenburg, Marco Roos, Adam Belloum, Dmitry Vasunin, Adianto Wibisono, Antoine van Kampen, and Timo Breit. Sigwin-detector: a grid-enabled workflow for discovering enriched windows of genomic features related to dna sequences. *BMC Research Notes*, 1(1):63, 2008.

[37] Thomas C. Hudson Ann E. Stapleton Ronald J. Vetter Tristan Carland Andrew Martin Jerry Martin Allen Rawls William J. Shipman Jeffrey L. Brown, Clayton S. Ferner and Michael Wood. GridNexus: A Grid Services Scientific Workflow System. In *International Journal of Computer Information Science (IJCIS), Vol 6, No 2*, pages 72–82, June 20 2005.

[38] Ian J.Taylor and Andrew Harrison. *From P2P and Grids to Web Services on the Web*. Springer, 2 edition edition, 2009.

[39] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[40] V. Korkhov, A. Wibisono, D. Vasyunin, and A.S.Z. Belloum et al. Vlam-g: Interactive data driven workflow engine for grid-enabled resources. *Scientific Programming*, 15(3):173–188, 2007.

[41] Vladimir Korkhov, Dmitry Vasyunin, Adianto Wibisono, Victor Guevara-Masis, Adam Belloum, Cees de Laat, Pieter Adriaans, and L.O. Hertzberger. Ws-vlam: towards a scalable workflow system on the grid. In *WORKS '07: Proceedings of the 2nd workshop on Workflows in support of large-scale science*, pages 63–68, New York, NY, USA, 2007. ACM.

[42] Edward A. Lee and Thomas Parks. Dataflow process networks. In *Proceedings of the IEEE*, pages 773–799, 1995.

[43] Yi Lin, Bettina Kemme, Marta Patio-martnez, and Ricardo Jimnez-peris. Enhancing edge computing with database replication.

[44] Bertram Ludäscher, Ilkay Altintas, Shawn Bowers, Julian Cummings, Terence Critchlow, Ewa Deelman, David De Roure, Juliana Freire, Carole Goble, Matthew Jones, Scott Klasky, Timothy McPhillips, Norbert Podhorszki, Claudio Silva, Ian Taylor, and Mladen Vouk. Scientific process automation and workflow management. In Arie Shoshani and Doron Rotem, editors, *Scientific Data Management*, Computational Science Series, chapter 13. Chapman & Hall, 2009.

[45] T Maeno. Panda: distributed production and distributed analysis system for atlas. *Journal of Physics: Conference Series*, 119(6):062036 (4pp), 2008.

[46] Elton Mathias, Françoise Baude, and Vincent Cave. A gcm-based runtime support for parallel grid applications. In *CBHPC '08: Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance*, pages 1–10, New York, NY, USA, 2008. ACM.

[47] Anne H. H. Ngu, Shawn Bowers, Nicholas Haasch, Timothy M. McPhillips, and Terence Critchlow. Flexible scientific workflow modeling using frames, templates, and dynamic embedding. In Bertram Ludscher and Nikos Mamoulis, editors, *SSDBM*, volume 5069 of *Lecture Notes in Computer Science*, pages 566–572. Springer, 2008.

[48] P Nilsson. Experience from a pilot based system for atlas. *Journal of Physics: Conference Series*, 119(6):062038 (6pp), 2008.

[49] James Norton. Dynamic class loading for C++ on Linux. `http://www.linuxjournal.com/article/3687`, May 2000. from Linux Journal issue 73.

[50] T. Oinn, M. Greenwood, M. J. Addis, M. Nedim Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. J. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: Lessons in creating a workflow environment for the life sciences. *JOURNAL OF CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE*, 2002.

[51] Michael A. Palis, Jing-Chiou Liou, and David S.L. Wei. Task clustering and scheduling for distributed memory parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 7:46–55, 1996.

[52] S K Paterson and A Tsaregorodtsev. Dirac optimized workload management. *Journal of Physics: Conference Series*, 119(6):062040 (9pp), 2008.

[53] Jun Qin, Thomas Fahringer, and Sabri Pllana. UML Based Grid Workflow Modeling under ASKALON. In *Proceedings of 6th Austrian-Hungarian Workshop on Distributed and Parallel Systems*, Innsbruck, Austria, September 21-23 2006. Springer-Verlag.

[54] I Sfiligoi. glideinwms—a generic pilot-based workload management system. *Journal of Physics: Conference Series*, 119(6):062044 (9pp), 2008.

[55] Warren Smith, Ian Foster, and Valerie Taylor. Scheduling with advanced reservations. *IEEE Int. Par. and*, 2000.

[56] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[57] Ian Taylor. Triana generations. In *E-SCIENCE '06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, page 143, Washington, DC, USA, 2006. IEEE Computer Society.

[58] Ian Taylor, Ian Wang, Matthew Shields, and Shalil Majithia. Distributed computing with triana on the grid: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(9):1197–1214, 2005.

[59] A Tsaregorodtsev, M Bargiotti, N Brook, A C Ramo, G Castellani, P Charpentier, C Cioffi, J Closier, R G Diaz, G Kuznetsov, Y Y Li, R Nandakumar, S Paterson, R Santinelli, A C Smith, M S Miguelez, and S G Jimenez. Dirac: a community grid solution. *Journal of Physics: Conference Series*, 119(6):062048 (12pp), 2008.

[60] Gregor von Laszewski and Mihael Hategan. Workflow Concepts of the Java CoG Kit. *J. Grid Comput.*, 3(3-4):239–258, 2005.

[61] Jianwu Wang, Ilkay Altintas, Parviez R. Hosseini, Derik Barseghian, Daniel Crawl, Chad Berkley, and Matthew B. Jones. Accelerating parameter sweep workflows by utilizing ad-hoc network computing resources: An ecological example. *Services, IEEE Congress on*, 0:267–274, 2009.

[62] George Wells. Coordination languages: Back to the future with linda. In *Proceedings of WCAT05*, pages 87–98, 2005.

[63] Adianto Wibisono, Dmitry Vasunin, Vladimir Korkhov, Zhiming Zhao, Adam Belloum, Cees de Laat, Pieter W. Adriaans, and Bob Hertzberger. WS-VLAM: A GT4 based workflow management system. In Yong Shi, G. Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *Computational Science - ICCS 2007, 7th International Conference, Beijing, China, May 27 - 30, 2007, Proceedings, Part III*, volume 4489 of *Lecture Notes in Computer Science*, pages 191–198. Springer, 2007.

[64] Marek Wieczorek, Radu Prodan, and Thomas Fahringer. Scheduling of scientific workflows in the askalon grid environment. *SIGMOD Rec.*, 34(3):56–62, 2005.

[65] Jia Yu and Rajkumar Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34(3):44–49, 2005.

[66] Jia Yu and Rajkumar Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *J. Grid Comput*, 3(3-4):171–200, 2005.

[67] Jia Yu and Rajkumar Buyya. Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. *Sci. Program.*, 14(3,4):217–230, 2006.

[68] Daniel Zinn, Shawn Bowers, Timothy McPhillips, and Bertram Ludäscher. Scientific workflow design with data assembly lines. In *WORKS '09: Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, pages 1–10, New York, NY, USA, 2009. ACM.