Prediction-based auto-scaling of scientific workflows

Cushing, R.; Koulouzis, S.; Belloum, A.S.Z.; Bubak, M.

DOI
10.1145/2089002.2089003

Publication date
2011

Document Version
Final published version

Published in
Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science: MGC 2011, co-located with ACM/IFIP/USENIX 12th International Middleware Conference, December 12-16, 2011, Lisbon, Portugal

Prediction-based Auto-scaling of Scientific Workflows

Reginald Cushing
Informatics Institute
University of Amsterdam
R.S.Cushing@uva.nl

Spiros Koulouzis
Informatics Institute
University of Amsterdam
S.Koulouzis@uva.nl

Adam S. Z. Belloum
Informatics Institute
University of Amsterdam
A.S.Z.Belloum@uva.nl

Marian Bubak
Informatics Institute
University of Amsterdam
and AGH Krakow
M.T.Bubak@uva.nl

ABSTRACT

In this paper we propose a novel method for auto-scaling data-centric workflow tasks. Scaling is achieved through a prediction mechanism where the input data load on each task within a workflow is used to compute the estimated task execution time. Through load prediction, the framework can take informed decisions on scaling multiple workflow tasks independently to improve overall throughput and reduce workflow bottlenecks. This method was implemented in the WS-VLAM workflow system and with an image analyses workflow we show that this technique achieves faster data processing rates and reduces overall workflow makespan.

Keywords

workflows, dataflow, auto-scaling, replication, messaging, message pipeline

1. INTRODUCTION

Scientific Workflow Management Systems (SWMS) have emerged as ubiquitous tools for eScience applications. Such tools aid scientists to compose complex scientific experiments that can be scheduled and executed on distributed systems such as grids and cloud e-infrastructures. As eScience is increasingly becoming data-centric, SWMS tools are nowadays faced with challenges as how to deal with massive datasets such as data processing and data transport. This shift towards data in eScience has given rise to the fourth paradigm in scientific discoveries [8] where it is envisioned that data analyses will play a central role in future discoveries. Tools such as SWMSs can play a vital role in accelerating discoveries by providing means for coordination of data-centric workflows where computation can automatically scale to meet the data demands.

Auto-scaling deals with automatically replicating tasks within a workflow as a means to speedup data processing rates. Workflows are commonly described as Directed Acyclic Graphs (DAG) where vertices represent computation tasks while edges represent dependencies between tasks. In SWMSs such as WS-VLAM [9], tasks also include a list of input and output data ports which, apart from the data dependency, also describe data channels between tasks. Tasks within a data-centric scientific workflow are often data dependent on each other where each task can, potentially, be a data intensive task. Managing multiple data-intensive tasks in SWMS poses a coordination challenge since the progress of the whole workflow is easily hampered by the slowest task. Data-centric tasks follow a pattern of consuming data chunks, processing the data and output results. This cycle is repetitive and the amount of data chunks that need to be processed directly influences the execution time of the task. A prediction engine for each task can infer the load on a task by monitoring the queued data chunks for a given task and computing the estimated execution time using heuristics from previous data processing times. Having a high load on a task, a coordinator can decide to replicate the task and partition the data chunks amongst them thus reducing the apparent load since data is consumed at a higher rate. Within scientific workflows, where each task is subjected to this prediction engine, each task can independently scale to accelerate its consumption rate and match its predecessor's data production rate thus maintaining a steady flow of data through the workflow.

A common pattern in scientific applications for achieving higher throughput by partitioning data amongst identical tasks is using a master/slave model [11]. In such a model, a master coordinator is responsible for disseminating data chunks to all slaves. This approach does not usually consider auto-scaling the amount of slaves and most often relies on over-provisioning resources by, greedily, initiating as many salves as possible. Such an approach is not well suited for scientific workflows since each task can possibly hog all the available resources by initiating many slaves and hence starve the rest of the workflow which will impede its progression.

In this paper we describe a new method for independently auto-scaling data-centric workflow tasks using a prediction-based approach, its implementation is called Datafluo. Size of queued data awaiting to be processed is used as an indicator for the task load. The role of the SWMS is to reduce each task load by replicating tasks so as to maintain workflow progression and reduce overall makespan. We also shows how scientific logic can be abstracted from the underlying distributed architecture intricacies such as communication and data transport through a method of task harnessing.

The paper is organized as follows: section 2 is an overview of work related to our contribution. Section 3 describes the model of computation for orchestrating workflows and achieving auto-scaling. Section 4 describes the implementation of the Datafluo architecture. Section 5 evaluates the architecture using a typical scientific workflow. Section 6 discusses future work and concludes.

2. RELATED WORK

Our work on auto-scaling is closely related to the master/slave patterns which are commonly used within the scientific community to scale up parameter study type applications. MapReduce [6, 3] is a recent popular programming models that follows the master/slave approach. In a typical master/slave execution pattern, a master program acts as the coordinator for a set of slaves by assigning work to them. In this case MapReduce works in a push model whereby a master coordinates the whole execution of its slaves by disseminating work chunks to slave nodes. This entails that the master needs to keep track of the work chunk distribution. Optimizations of this model includes better scheduling of data chunks to heterogeneous resources [15] where the size of the data chunks assigned to a node is proportional to the node performance.

Combining the MapReduce model within a scientific workflow engine has been done in Kepler [16] which represents the mapping and reduce functions as separate tasks within a workflow. In [16], Hadoop is integrated with Kepler hence the work dissemination back-end still depends on a master coordinator to manage a set of slaves. In [5] a MapReduce high performance workflow system for GIS is proposed. This system estimates the number of resources needed to perform a particular distributed execution by simulation using user-defined estimated task execution.

For handling better task manageability during runtime, tasks are commonly submitted to grids using late binding mechanisms. In late binding, pilot jobs such as Condor Glideins [12] are submitted as place holders instead of the actual job or user-overlay systems like GANGA/DIANE [11] are applied. Once the place holder is executed on a node, the actual job is pulled into the node for execution. This late binding technique circumvents scheduling queueing waiting time. Pilot job mechanisms lack thorough task management such as abstracting the scientific logic from the underlying communication and data transport complexities.

3. MODEL OF TASK AUTO-SCALING

Auto-scaling targets workflows that are represented as DAGs with vertices representing tasks. Tasks, in turn, have a list of its input and output ports. Edges in the DAG represent data communication channels and thus, the data dependency between tasks.

We model a workflow W as a set of interdependent dataflow tasks $\{t_1, t_2, ..., t_n\}$ which are matched to the set of resources R. Tasks are represented as tuples

$$< id, st, IP, OP, PT, DT, IC, OC >$$

where id is the task id, st is the allocated computing slot time for a given task, IP is the set of input ports, OP is the set of output ports, PT is the set of tasks that precede task t_{id} where $PT \subset W$, DT is the set of dependent tasks that follow task t_{id} where $DT \subset W$. IC is the set of input data channels between output ports of tasks in set PT to input ports for task t_{id}. Similarly, OC is the set of output data channels between output ports for task t_{id} to input ports of tasks in DT. Ports consume and produce a set of messages $\{m_1, m_2, ..., m_n\}$, messages are consumed and produced sequentially. The dataflow model dictates that a task t_k will only be matched to a resource in R when, for each input port IP_{t_k}, the first message m_1 is delivered.

Auto-scaling is achieved by monitoring messages between tasks, the auto-scaling module can deduce which tasks are overloaded by predicting the completion time. Given a task t_k, replication can be applied by monitoring a designated port $ip_{t_k} \in IP_{t_k}$. The first step in auto-scaling is to keep track of the data processing rate of t_k on a node $r_j \in R$. $M_{ip.t_k}$ is the set of messages for the designated port ip_{t_k}. $C_{ip.t_k} \subset M_{ip.t_k}$ is the set of messages already consumed by ip_{t_k} where the current message being processed is also part of this set. $Q_{ip.t_k} \subset M_{ip.t_k}$ is the set of messages yet to be consumed by ip_{t_k}. The function $time_{ip.t_k}(m_j)$ records the time a message has been delivered to input port ip_{t_k}. The processing time of a message is defined as the interval time recorded between successive messages thus $mit_{ip.t_k}(m_{l-1}, m_l) = time_{ip.t_k}(m_l) - time_{ip.t_k}(m_{l-1})$. $size(m_k)$ represents the data size of m_k thus the actual data size of the set M, $size(M) = \sum_{j=1}^{n} size(m_j)$. Since messages can have different data sizes, auto-scaling module needs to calculate the data processing rate. This is done on every message being consumed by a port and is defined as:

$$proc(ip_{t_k}) = \frac{size(C_{ip.t_k})}{\sum_{j=2}^{n} mit_{ip.t_k}(m_{j-1}, m_j)},\ n = |C_{ip.t_k}|. \quad (1)$$

Equation 1 calculates the data processing rate in bytes per second and this includes also the overhead of delivering a message, the overhead to start processing the next message, and depending on the implementation of the task, the calculation, would typically include the time for producing messages on output ports as a response to processing input messages. Equation 2 calculates the expected time for completion.

$$pred(ip_{t_k}) = size(Q_{ip.t_k}) \times proc(ip_{t_k}). \quad (2)$$

The prediction is calculated on every consumed message and is averaged out with the last calculated prediction to smooth out large spikes in the prediction graph. Task t_k is said to be overloaded if for the designated port ip_{t_k}, $pred(ip_{t_k}) > t_k^{st}$ alternatively, t_k^{st} could be substituted by a user-defined threshold. A simple calculation of the needed number of clones to reduce the completion time below t^{st} is

$$repl(t_k) = \frac{pred(ip_{t_k})}{t_k^{st}}. \quad (3)$$

Equation 3 assumes that all messages have approximately the same size which may not always be the case thus another solution is to factor in the standard deviation from the mean message size such that

$$corr(Q_{ip.t_k}) = 1 - \frac{mean(Q_{ip.t_k}) - stdd(Q_{ip.t_k})}{mean(Q_{ip.t_k})}. \quad (4)$$

The final number of clone to be initiated can then be calculated as $ceil(repl(t_k) \times corr(Q_{ip.t_k}))$. The greater the message size standard deviation the less clones are initiated. Large standard deviation results in messages having drastic variation in their data sizes and can lead to over-provisioning resources through inaccurate clone number calculation.

Another source of variation in the number of clone calculation is the fact that the prediction is based on some resource with its own characteristics CPU power and memory capacity. Since resources in distributed systems are intrinsically heterogeneous, the time to process messages on one resources might not be the same as on other resources. This may lead to over-provisioning when clone number estimation is done on a relatively slow resource.

To cater for over-provisioning, clones are scheduled in bursts. When a burst of clones is scheduled, auto-scaling continues predicting the estimated completion time which would now include the data processing of all instances of the replicated task. If the task is still overloaded more bursts are scheduled until the load is reduced within the acceptable limit.

4. IMPLEMENTATION OF AUTO-SCALING

4.1 Architecture of the Datafluo System

Figure 1 depicts a high-level overview of the Datafluo system architecture. The system is composed of a set of loosely coupled modules bound by a central messaging back-end. The Datafluo system implements a two-step scheduling strategy. The Datafluo enactment engine represents the top-level scheduler which models the workflow task data dependencies. The enactment engine deals with tasks at an abstract level and merely marks tasks as runnable when their dependencies are met. The bottom-level scheduler deals with scheduling tasks on a set of resources thus its main role is matchmaking. The central module in the whole system is the messaging back-end which binds all the modules together.

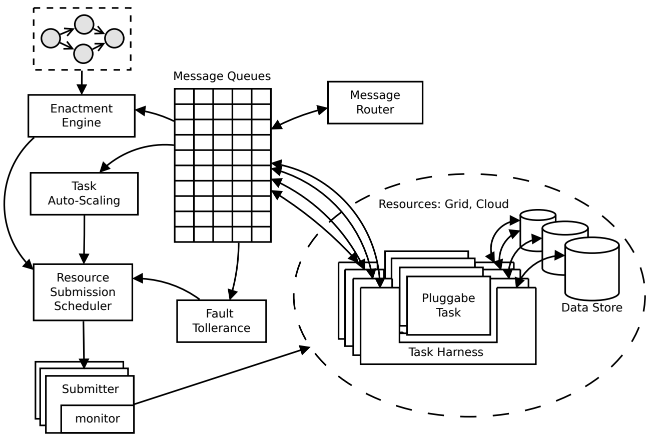

Figure 1: Loosely coupled Datafluo system modules revolving around a core messaging module

The Datafluo enactment engine is the entry point into the system. It accepts a WS-VLAM [10] DAG workflow generated by a workflow composer. At this stage the workflow is interpreted and a dataflow object representation is generated.

When a task is made runnable (i.e. all input ports have data) it is passed to the bottom-level scheduler for matching to a resource. The architecture allows for different schedulers to be implemented. Default schedulers are *round-robin* which circularly matches tasks to resources and therefore achieves load balancing between resources, *bucket* scheduler which orders resources by the amount of slots available and fills up the resources consecutively starting from the largest resource thus achieving locality between tasks, *cloud* scheduler which takes into consideration a budget for running a workflow and elastically expands the resource pool R by calling an interface to cloud resources [14] for on-demand cluster creation to accommodate more tasks.

4.2 Data Queueing

The message queues play a pivotal role in the whole architecture. Most importantly message queues allows inter-task communication over computing infrastructure domains which, most often, have restricted Internet access. Intermediate messages allow tasks to exploit fine concurrency between dependent tasks. As with message streams, the granularity of concurrency depends on the task logic and how often messages are produced and consumed. A one-to-one mapping exists between the set of task input/output ports (IP, OP) and message queues. The message queues provide a persistent means of communication between tasks which in turn decouples task execution in time and hence eliminates the need to co-allocate resources. Co-allocations is known to degrade the system due to increased task waiting times [7, 13]. As depicted in Figure 1, the message queuing also allows for modules such as the message router and the enactment engine to spoof on the messages being transmitted. Based on the message routing, the enactment engine can infer which tasks have data on their ports and thus can make tasks runnable.

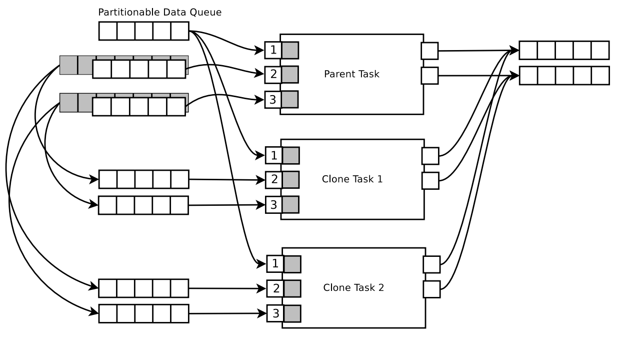

Figure 2: Queue setup strategy for achieving auto-scaling. Parent non partition-able data queues have associated shadow queues which enable clones to retrieve the whole input message set any time. Input ports also have a reserve port which is used by the fault tolerance subsystem to replay the last message in case the task is re-submitted.

Figure 2 depicts the queuing strategies that supports cloning tasks. Cloning is the procedure of replicating a task by the scaling subsystem. The parent task is responsible for managing its own clone farm which means that the Datafluo enactment engine has no knowledge of replication taking place. This preserves the original workflow semantics. Port 1 of the parent task is the designated port for which auto-scaling will

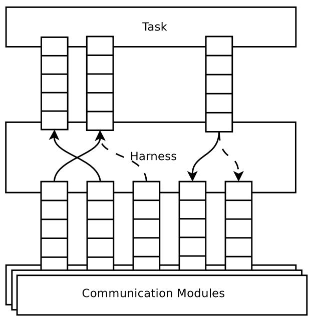

Figure 3: Harness setup for a task with 2 input ports and 1 output port

perform prediction and replication. All instances of the same task will share the designated port and thus the data is partitioned amongst all clones. Ports 2 and 3 are not partitioned amongst clones. This is due to the complexity and ambiguity of how to partition multiple ports amongst a set of clones. The system guarantees that for any non-designated input ports, all clones have the same input message set therefore in Figure 2 the input message sets M_{ip_2} and M_{ip_3} are identical to the parent and clones. This is achieved through shadow queues on the central message exchange. A shadow queue acts as a buffer for the set of messages consumed by the parent. Clone input ports are attached to the shadow queues instead of the standard queues. All clones share the same output ports with the parent. By default no ordering is done on the message output queues hence messages are delivered out of order. Ordering is an expensive routine and can be achieved through the message sequence numbers.

4.3 Task Harnessing

The unit of submission is a task harness. The main goals of the task harness are: it allows task late binding by dynamically plugging tasks and abstracts the underlying data management and communication from the core scientific logic. The task harness is responsible for retrieving messages from the queues, interpreting the protocol used in the reference, loading the necessary communication libraries, retrieving the actual data from reliable storages, and pushing the data up to the task. On data output, the harness locates the closest data store from a list of stores, puts the data on the server and queues a message indicating the reference to the stored data.

The task harness architecture is based on plug-in model whereby the task and communication are dynamically loadable modules. The core of the harness is the data management fabric which binds loadable communication libraries to the task input/output ports through a system of queues. As shown in figure 3, these internal queues decouple the scientific logic from the underlying communication mechanisms. On starting the harness, the configuration is loaded which allows task late binding since it is only at this point that a task is assigned to a harness. On loading a task t_k, the harness sets up internal data queues for each task port in $IP_{tk} \cup OP_{t_k}$. Messages containing referenced data are handled by the harness by dynamically loading the appropriate protocol library, such as GridFTP, for retrieving the actual data. The communication library responds by pushing the data onto its internal queue (shown in the bottom part of Figure 3). The harness will then route the data from the communication queue to the relevant task input queue. Data output by the task happens in reverse order. When data is available on the tasks' internal output queue, the harness picks up the data, it then locates the closest data server from a list of servers, loads the required communication libraries and pushes the data to communication queue. The communication module picks up the data and sends it to the data server. The harness will then construct a message with the endpoint reference of the newly created data and sends it to the central message queue which is then routed to other tasks in the workflow.

Since communication in distributed environments can be quite restrictive due to security policies, the Datafluo system relies on a pull model whereby tasks initiate all communication. A pull model is the best guarantee that tasks can establish communication. The task harnesses poll the server for new messages. Polling implements an exponential back-off when no new messages are retrieved. When no messages are retrieved the polling interval is increased up until a fixed threshold or until a new message is retrieved. This reduces the load on the messaging back-end by not overwhelming the server with too many unnecessary requests.

5. EVALUATION

Figure 4 shows an example application using Octave[1] [2] aimed at illustrating the task level scaling implemented in the Datafluo system. The workflow has two starting points one being the *DirectoryReader* which, as the name suggests, reads images from a directory. The second starting task is the *Parameter* task. This task acts as a parametric engine which supplies parameters to the *Histogram* task. The *Normalize* task normalises RGB images. The tasks *Converter2* and *Converter1* convert images into different colour spaces. *Histogram* calculates the euclidean distance between the two new colour space histograms. At the end of the workflow the results are collected by *Results* while intermediate images are collected by *ImageCollector*. The core tasks of the workflow i.e. *Normalize*, *Converter2*, *Converter1*, and *Histogram* are embarrassingly parallel in nature. These tasks have no casual dependency between messages on the same port and therefore are ideal for a driving test case to test our Datafluo scaling systems.

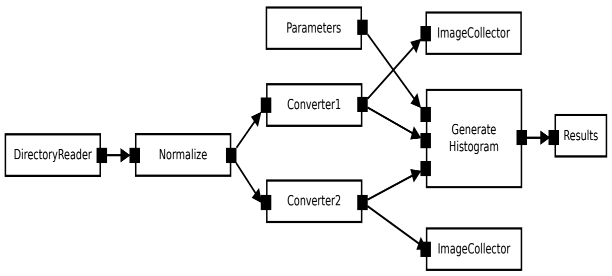

Figure 4: An Octave image processing workflow. The workflow converts images into two different colour spaces and calculates the histogram difference between the two new colour spaces. The parameter setting for the Histogram is the histogram bin size.

[1]GNU Octave is a language intended for numerical computations which is very similar to Matlab.

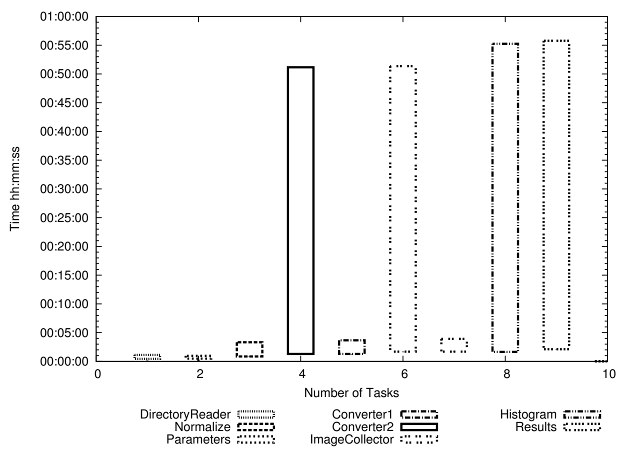

Figure 5: Workflow execution without scaling. The length of the bar represents the total execution time.

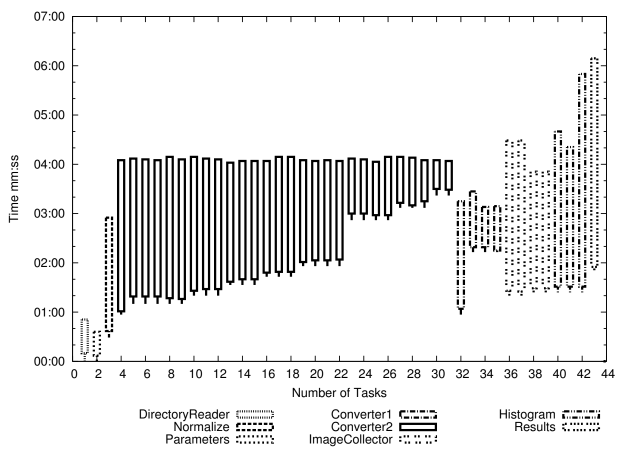

Figure 6: Workflow execution with scaling. Lines preceding the bars represent waiting time while the length of the bars represent actual execution time of an instance of the workflow task. Bars with the same line encoding are replicated instances of the same task. Time 0 represents the start waiting time for the first task.

As a resource pool back-end we had access to the Distributed ASCI SuperComputer 3 (DAS3) [1] which is a five wide area distributed system. Tasks where submitted across all clusters. Each site hosts a GridFTP server which where used to transmit data between tasks and therefore allow inter-cluster task communication. The timings illustrated in Fig. 5, Fig. 6 shows the execution time of each task in the workflow. The execution time incorporates the the scientific logic execution time as well as overheads associated with communication, startup and cleanup times. Since the DAS3 is a shared computing infrastructure, care was taken so that execution times where not influenced by other jobs running on DAS3. The experiment was executed 15 times. The execution profiles were deemed similar with minor deviations in the number of tasks replicated for *Converter 1* and *Converter 2*. This is due to the dynamism of the resources and the scheduling algorithm. The results depict one of the executions which is an ideal representative of all other samples.

Converter2 and *Converter1* are set to auto-scaling. In these cases the Datafluo system is responsible for gauging the load on the designated data partition input queue and decide on how many clones to submit using a user defined threshold. The *Parameter* task acts as the parameter engine by reading parameters from a file and sending messages containing parameters to the *Histogram.* The latter is not set to auto-scale but instead is scaled on a per parameter bases: each parameter from *Parameter* creates a new instance of *Histogram. ImageCollector* is set to a fixed replication where the user specifies the number of clones.

At time 0 *DirectoryReader* and *Parameters* are submitted as these have no dependencies. Other tasks are only submitted when some data is available for input. This is clearly shown by the difference in starting times for each task in Figure 6. The computation overlap between tasks shows the effect of message pipelining where tasks can start processing data immediately as it is produced and need not to wait for the dependant tasks to terminate before starting execution.

Figure 5 shows the workflow execution with scaling disabled. The results clearly show that *Converter2* is relatively slow to process the data and hence causes a flow bottleneck. This has a ripple effect on the dependant tasks (*Histogram*, *ImageCollector* and *Results*) which spend most of their time in an idling state waiting for new messages to be delivered to their input ports. Since scaling is completely disabled, the parameter sweep scenario does not take effect and hence *Histogram* is not replicated. This results in The *Histogram* task processing all parameters. The mean runtime for the non-scaled workflow is around 54 minutes.

Figure 6 shows the same example with the same inputs but this time enabling scaling features. The results immediately show how the previous bottleneck was circumvented through replication. The *Converter2* was replicated as many times as needed hence increasing the data consumption and production. The *Histogram* is replicated 3 times which follows the parameter sweep scenario whilst we have four *ImageCollectors* as defined by the user. All in all the 8 task workflow unfolded into 44 separate tasks through scaling. In this example the effect of just auto-scaling achieved a 9 fold improvement over the non scaled workflow. The single task *Converter2* achieves a much better improvement which is approximately 16 times faster. The execution profile for *Converter2* tasks also shows the burst threshold in action since tasks are replicated in bursts which gives rise to the staircase profile. From Figure 6, *Converter1* is also dynamically scaled up but since it is faster it has a lower replication count.

6. CONCLUSIONS AND FUTURE WORK

We have shown how prediction-based auto-scaling can be applied to data-centric workflows as a way to accelerate data processing rates within scientific workflows. The ability of scaling tasks independently enables replication of tasks to match the data production rate. This minimizes workflow bottlenecks and reduces total makespan. Through task harnessing we showed how scientific logic can be separated from underlying communication and data transport intricacies.

Although the system achieves auto-scaling, this is done through the SWMS which can result in a bottleneck to the system. An better approach is to shift most of the logic into the task harness. This leads to autonomous workflow orchestration where tasks can self scale and organise with minimal coordinator intervention.

Auto-scaling is an attractive approach especially within the context of scientific workflows where single tasks can independently scale themselves. Applications that can immediately benefit from this model belong to the class of data-centric applications with easily partitionable data. The example use-case demonstrated how improvements in execution time of a typical scientific workflow could be achieved through the proposed auto-scaling model. The architecture is not only limited to image analyses but can be equally applicable to other scientific fields. Ongoing research is under way to evaluate this system using a sequence alignment workflow from bio-informatics.

The evaluation of the Datafluo system is aimed at demonstrating the auto-scaling method as a way to speedup workflow execution. A further Datafluo system evaluation can illustrate the overhead incurred by the various modules of the system such as the Datafluo enactment engine, messaging, and task harness.

The modular structure of the Datafluo system makes is suitable for integrating with other workflow systems. Different workflow languages can be parsed by plugging in new parsers while legacy applications can be wrapped within standard task harnesses.

As part of further research, the Datafluo system is being extended to incorporate web services where the task harness is an actual modified service container. The modified service container will implement the harnessing logic and allow services to auto-scale and deployed on distributed systems. The pull model allows services to be invoked within network access restricted clusters while enabling service back-to-back communication through messaging.

7. ACKNOWLEDGMENTS

This research was partially funded by the COMMIT[2] and VPH-Share[3] projects.

8. REFERENCES

[1] Das3. `http://www.cs.vu.nl/das3`.

[2] Gnu octave. `http://www.gnu.org/software/octave`.

[3] Hadoop. `http://hadoop.apache.org`.

[4] A. Belloum, M. Inda, D. Vasunin, V. Korkhov, Z. Zhao, H. Rauwerda, T. Breit, M. Bubak, and L. Hertzberger. Collaborative e-science experiments and scientific workflows. *Internet Computing, IEEE*, 15(4):39 –47, july-aug. 2011.

[5] Q. Chen, L. Wang, and Z. Shang. MRGIS: A MapReduce-Enabled High Performance Workflow System for GIS. In *eScience '08: IEEE Fourth International Conference on eScience*, pages 646–651, Dec. 2008.

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.

[7] E. Elmroth, F. Hernández, and J. Tordsson. Three fundamental dimensions of scientific workflow interoperabilit:y model of computation, language, and execution environment. *Future Generation Computer Systems*, 26(2):245 – 256, 2010.

[8] T. Hey, S. Tansley, and K. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery.* Microsoft Research, Redmond, Washington, 2009.

[9] V. Korkhov, D. Vasyunin, A. Wibisono, V. Guevara-Masis, A. Belloum, C. de Laat, P. Adriaans, and L. 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.

[10] V. Korkhov, A. Wibisono, D. Vasyunin, and A. B. et al. VLAM-G: Interactive data driven workflow engine for Grid-enabled resources. *Scientific Programming*, 15(3):173–188, 2007.

[11] J. Moscicki, M. Lamanna, M. Bubak, and P. Sloot. Processing moldable tasks on the grid: Late job binding with lightweight user-level overlay. *Future Generation Computer Systems*, 27(6):725 – 736, 2011.

[12] I. Sfiligoi et al. The Pilot Way to Grid Resources Using glideinWMS. In *Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering - Volume 02*, pages 428–432, Washington, DC, USA, 2009. IEEE Computer Society.

[13] W. Smith, I. Foster, and V. Taylor. Scheduling with advanced reservations. *IEEE Int. Par. and*, 2000.

[14] R. Strijkers, W. Toorop, et al. Amos: Using the cloud for on-demand execution of e-science applications. In *Proceeding of the eScience2010*, pages 773–799, 2010.

[15] L. Thomas and B. Annappa. Utilization of map-reduce for parallelization of resource scheduling using MPI: PRS. In *Proceedings of the 2011 International Conference on Communication, Computing & Security*, ICCCS '11, pages 415–420, New York, NY, USA, 2011. ACM.

[16] J. Wang, D. Crawl, and I. Altintas. Kepler + Hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, WORKS '09, pages 12:1–12:8, New York, NY, USA, 2009. ACM.

[2]www.commit-nl.nl
[3]www.vph-share.eu