Kepler + Hadoop : A General Architecture Facilitating Data-Intensive Applications in Scientific Workflow Systems

Jianwu Wang, Daniel Crawl, Ilkay Altintas
San Diego Supercomputer Center, University of California, San Diego
9500 Gilman Drive, MC 0505
La Jolla, CA 92093-0505, U.S.A.
{jianwu, crawl, altintas}@sdsc.edu

ABSTRACT
MapReduce provides a parallel and scalable programming model for data-intensive business and scientific applications. MapReduce and its de facto open source project, called Hadoop, support parallel processing on large datasets with capabilities including automatic data partitioning and distribution, load balancing, and fault tolerance management. Meanwhile, scientific workflow management systems, e.g., Kepler, Taverna, Triana, and Pegasus, have demonstrated their ability to help domain scientists solve scientific problems by synthesizing different data and computing resources. By integrating Hadoop with Kepler, we provide an easy-to-use architecture that facilitates users to compose and execute MapReduce applications in Kepler scientific workflows. Our implementation demonstrates that many characteristics of scientific workflow management systems, e.g., graphical user interface and component reuse and sharing, are very complementary to those of MapReduce. Using the presented Hadoop components in Kepler, scientists can easily utilize MapReduce in their domain-specific problems and connect them with other tasks in a workflow through the Kepler graphical user interface. We validate the feasibility of our approach via a word count use case.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *Distributed Applications*; D.2.11 [**Software Engineering**]: Software Architectures – *Domain-specific architectures*.

General Terms
Design, Experimentation, Performance

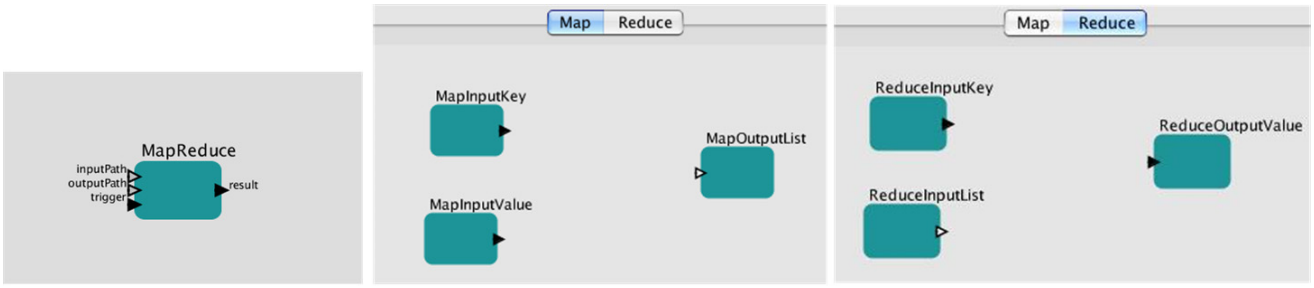
Keywords
MapReduce, Kepler, Hadoop, scientific workflow, parallel computing, distributed computing, easy-to-use.

WORKS 09, November 15, 2009, Portland Oregon, USA.

1. INTRODUCTION
MapReduce [1] provides a parallel and scalable programming model for data-intensive business and scientific analysis. Since 2003, MapReduce and the open source Hadoop [2] platform based on MapReduce, have been successfully and widely used on many business applications that require parallel processing on large datasets. Currently, more and more large-scale scientific problems are facing similar processing challenges on large scientific datasets [3], where MapReduce could potentially help [4, 5]. There has been some work utilizing MapReduce and Hadoop for scientific problems such as CloudBurst algorithm [6] in bioinformatics and MRGIS framework [7] in geoinformatics. However, there are still two main difficulties that make it hard for domain scientists to benefit from this powerful programming model. First, the application logic in MapReduce usually needs to be expressed in traditional programming languages, such as Java or shell scripts. It is not trivial for scientists to learn the MapReduce API and write corresponding programs. Second, only a part of the solution for a scientific problem is usually suitable to utilize MapReduce. The MapReduce framework needs to be easily integrated with other domain applications that do not require MapReduce.

The contribution of this paper alleviates these difficulties through a domain-independent and easy-to-use architecture that enables the usage of MapReduce in scientific workflows by integrating Hadoop with Kepler [8]. Through Kepler, a user-friendly open source scientific workflow system, users can easily utilize MapReduce in their domain-specific problems and connect them with other tasks in a workflow through a graphical user interface. Besides application composition and execution, the architecture also supports easy reuse and sharing of domain-specific MapReduce components through the Kepler infrastructure.

The paper is organized as follows. In Section 2, we describe Kepler, MapReduce and Hadoop. The details of our architecture are explained in Sections 3. Section 4 describes the application of our architecture for a widely used word count use case. We compare our work with related work in Section 5. Finally, we conclude and explain future work in Section 6.



(a) (b) (c)

Figure 1. (a) MapReduce actor. (b) Map sub-workflow in MapReduce actor. (c) Reduce sub-workflow in MapReduce actor.

2. BACKGROUND

2.1 Kepler

The Kepler project[1] aims to produce an open source scientific workflow system that allows scientists to design and efficiently execute scientific workflows. Since 2003, Kepler has been used as a workflow system within over 20 diverse projects and multiple disciplines.

Inherited from Ptolemy II[2], Kepler adopts the *actor-oriented modeling* [8] paradigm for scientific workflow design and execution. Each actor is designed to perform a specific independent task that can be implemented as *atomic* or *composite*. Composite actors, or *sub-workflows*, are composed of atomic actors bundled together to perform complex operations. Actors in a workflow can contain *ports* to consume or produce data, called *tokens*, and communicate with other actors in the workflow through communication channels via *links*.

Another unique property inherited from Ptolemy II is that the order of execution of actors in the workflow is specified by an independent entity called the *director*. The director defines how actors are executed and how they communicate with each other. The execution model defined by the director is called the *Model of Computation* [8]. Since the director is decoupled from the workflow structure, a user can easily change the computational model by replacing the director using the Kepler graphical user interface. As a consequence, a workflow can execute either in a sequential manner, e.g., using the Synchronous Data Flow (SDF) director, or in a parallel manner, e.g., using the Process Network (PN) director.

Kepler provides an intuitive graphical user interface and an execution engine to help scientists to edit and manage scientific workflows and their execution. The execution engine can be separated from the user interface enabling the execution of existing workflows in batch mode. Actors are dragged and dropped onto Kepler canvas, where they can be customized, linked and executed. With built-in wizard tools, customized actors can be easily exported for reuse and sharing locally or publicly through the Kepler actor repository[3].

2.2 MapReduce and Hadoop

The MapReduce [1] is a parallel and scalable programming model for data-intensive computing, where input data is automatically partitioned onto multiple nodes and user programs are distributed and executed in parallel on the partitioned data blocks. It consists of two functions as shown in Table 1: Map function processes on a portion of the whole data set and produces a set of intermediate key-value pairs, and Reduce function accepts and merges the intermediate pairs generated from Map. MapReduce supports data partitioning, scheduling, load balancing, and fault tolerance. Following the simple interface of MapReduce, programmers can easily implement parallel applications.

Table 1. MapReduce Programming model [1]

map (k1,v1) → *list(k2,v2)*
reduce (k2,list(v2)) → *list(v2)*

The Hadoop project [2] provides an open source implementation of MapReduce. It is composed of a MapReduce runtime system and a distributed file system, called HDFS. HDFS supports MapReduce execution with the capability of automatic data redundancy and diffusion among each node in the Hadoop cluster. Hadoop also handles node failures automatically. One Hadoop node, called *master*, dispatches tasks and manages the executions of the other Hadoop nodes, i.e., *slaves*.

3. KEPLER + HADOOP ARCHITECTURE

The Kepler + Hadoop architecture enables Kepler users to easily express and efficiently execute their domain-specific analyses with MapReduce. We explain this architecture in the following sub-sections.

3.1 MapReduce Actor in Kepler

A new composite actor called *MapReduce* is implemented in Kepler to provide a graphical user interface for expressing Map and Reduce logic. The MapReduce actor can then be placed in a larger workflow to connect the MapReduce functionality with the rest of the workflow. Since Map and Reduce are two separate functions, we treat Map and Reduce as two independent sub-workflows in our MapReduce actor. The MapReduce actor and the separate Map and Reduce tabs for the two sub-workflows are shown in Figure 1. According to the interfaces of Map and Reduce in Table 1, we implement corresponding auxiliary actors that transfer their inputs and outputs. For instance, the Map function reads *k1* and *v1* as inputs and generates *list* (*k2, v2*). Corresponding actors, called MapInputKey, MapInputValue and MapOutputList are provided in Kepler. When users drag and drop the MapReduce actor onto Kepler canvas, the Map and Reduce sub-workflows are inside of it with the auxiliary input/output

[1] http://www.kepler-project.org/

[2] http://ptolemy.eecs.berkeley.edu/ptolemyII/

[3] http://library.kepler-project.org/kepler/

actors by default, showing as two separate tabs of the MapReduce actor. Users of the MapReduce actor only need to complete the two sub-workflows by dragging and dropping other Kepler actors to create sub-workflows that solve their domain-specific problems. The process is the same with that of constructing normal Kepler workflows. In the end, each MapReduce actor will have two separate sub-workflows containing Map and Reduce logic.

As shown in Figure 1, the MapReduce actor has three input ports, namely *trigger*, *inputPath* and *outputPath*, and one output port, called *result*. The inputPath and outputPath ports specify the paths of input and output data. During the initialization phase of the MapReduce actor, input data can be transferred from the local file system to HDFS. This allows data dynamically produced by upstream actors in the workflow to be used by the MapReduce actor. Alternatively, the MapReduce actor can be configured to use input data already in HDFS, which fits the cases where input data is static and very large. The output data can also be optionally copied to the local file system via the outputPath port. The trigger port provides a control signal to run the actor; users may need other conditions for actor execution in addition to the availability of input data. The result port shows the success or failure of the MapReduce actor, which can be used to trigger downstream actors.

As with any actor in Kepler, the MapReduce actor can be connected to other actors through its ports so that the tasks outside of the MapReduce programming model can be easily integrated in Kepler. This fits well with the characteristics of many scientific analyses that are composed of many tasks but only parts of them are suitable to utilize MapReduce.

Users can also easily create domain-specific MapReduce actors for reuse and sharing. Once the Map and Reduce sub-workflows are created, a user can save them locally for private sharing or upload them to the centralized Kepler actor repository for public sharing. We will show a customized actor for a word count case in Section 4. Additional MapReduce actors are being studied to support complex domain-specific functions, such as CloudBurst algorithm in the bioinformatics domain [6], which uses MapReduce for highly sensitive read mapping, and read mapping spatial stochastic birth-death process simulation in the ecological domain [9, 10], , which could utilize MapReduce for parallel execution of parameter sweep applications.

3.2 MapReduce Actor Usage

In this subsection we analyze the usage of our MapReduce actor for different user roles, and its corresponding benefits, which is illustrated in Figure 2.

Actor developers understand the MapReduce programming model and know which domain-specific tasks can benefit from this model, so they can create domain-specific (also called customized) MapReduce actors by completing the Map and Reduce sub-workflows in Figure 1. Each domain-specific MapReduce actor will perform a certain independent function, such as the MapReduce4WordCount actor will count the occurrence number of each word in input files, and the MapReduce4CloudBurst actor will implement highly sensitive read mapping in bioinformatics. All this work can be done via Kepler GUI without writing a single program line.

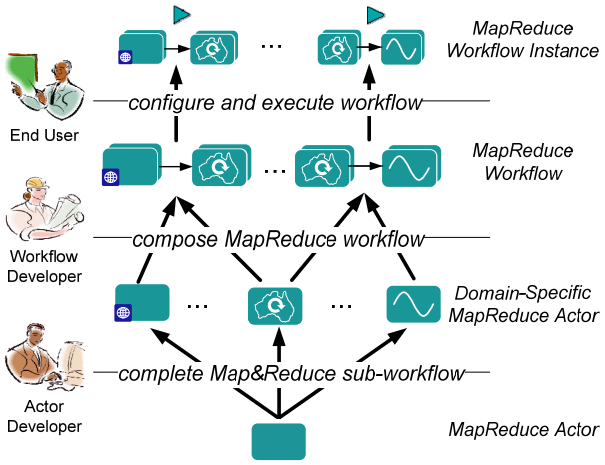


Figure 2: The usage of MapReduce actor.

Workflow developers know the complete domain-specific requirements, so they can choose the needed domain-specific MapReduce actors from the actor library, and connect them with other necessary actors (e.g., actors for data pre-processing and post-processing). By reusing domain-specific MapReduce actors pre-defined by actor developers, workflow developers can easily compose workflows without knowing the MapReduce programming model.

Since end users are acquainted with their concrete workflow configurations and execution environments, they can choose needed workflow from workflow library, configure actors in the workflow (such as providing input data and specifying workflow parameter values), and execute the whole workflow through the Kepler GUI or batch mode on target computing resources. By configuring and executing pre-constructed workflows, end users can easily fulfill their specific requirements without workflow composition knowledge.

3.3 MapReduce Actor Execution in Hadoop

As explained above, the MapReduce composite actor provides a capability for actor developers to express Map and Reduce logic as a part of their domain-specific workflows. The Hadoop framework provides powerful features such as data partitioning, parallel execution, load balancing, and fault tolerance. In this sub-section, we will discuss how to combine the above efforts to achieve efficient execution of the MapReduce actor in Hadoop.

The whole architecture is shown in Figure 3, which consists of three layers: the Kepler GUI, the Hadoop Master, and Map and Reduce Slaves. One key principle of the MapReduce programming model is moving computation rather than moving data[4]. So during the initialization of the MapReduce actor, the input data stored in the local file system will be first transferred to HDFS, which will then automatically partition and distribute the data to Map Slaves. After the data stage-in phase, the Kepler execution engine and the Map/Reduce sub-workflows in the MapReduce actor will be distributed to slaves for Map/Reduce tasks. The Map/Reduce sub-workflows will be executed with the data blocks on the slaves. Throughout execution, Hadoop will

4 http://hadoop.apache.org/common/docs/current/hdfs_design.html

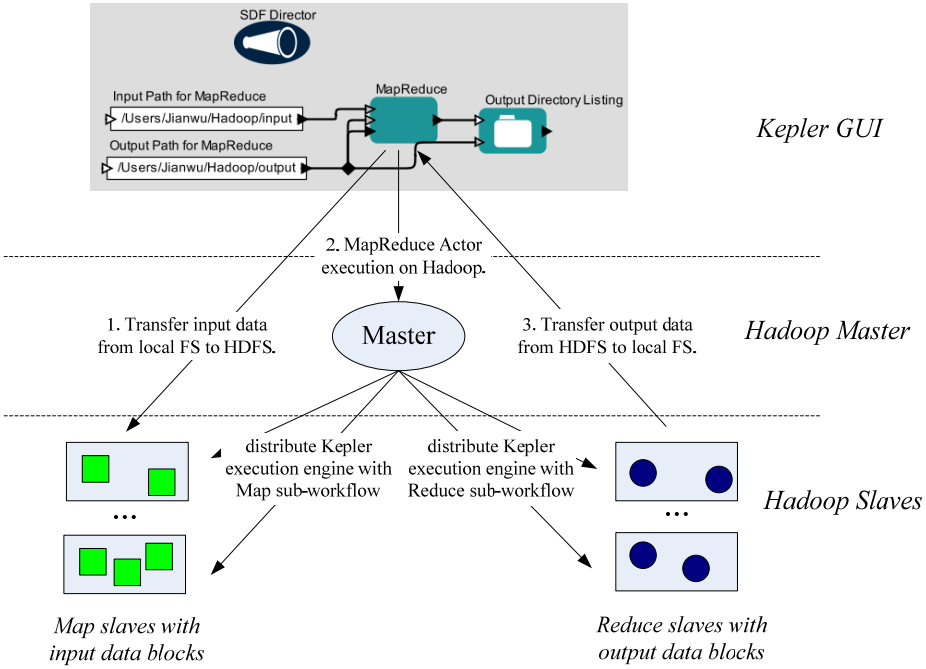


Figure 3: Architecture for MapReduce actor execution in Hadoop.

provide fault tolerance through slave monitoring and data replication mechanisms. After execution completes on the Hadoop slaves, the MapReduce actor will automatically transfer output data from HDFS to the path in the local file system specified in its outputPath port. As discussed in sub-section 3.1, users can also manually stage-in data before workflow execution and stage-out data after execution in order to save workflow execution time.

Table 2. Execution semantics in Map and Reduce function for Kepler MapReduce actor

```
map (k1, v1) {
  initialize Kepler execution engine for Map sub-workflow
  send k1 to Kepler engine via MapInputKey actor
  send v1 to Kepler engine via MapInputValue actor
  execute Map sub-workflow
  get list(k2, v2) from Kepler engine via MapOutputList actor
  emit  list(k2, v2)
}

reduce (k2, list(v2)) {
  initialize Kepler execution engine for Reduce sub-workflow
  send k2 to Kepler engine via ReduceInputKey actor
  send list(v2) to Kepler engine via ReduceInputList actor
  execute Reduce sub-workflow
  get v2 from Kepler engine via ReduceOutputValue actor
  emit (k2, v2)
}
```

We implemented the Map and Reduce interface provided by Hadoop. When execution begins, the input data read by the Hadoop slaves will be transferred to the Map and Reduce sub-workflows by our auxiliary input actors, such as the MapInputKey and MapInputValue actor. Next, the Kepler engine will execute the Map/Reduce sub-workflows with the input data. Finally, our auxiliary output actors will transfer the output data of the sub-workflows to the Hadoop slaves. The execution semantics for MapReduce actor execution in the Map and Reduce function are illustrated in Table 2.

Our architecture provides a generic interface by which any MapReduce application can be created and executed. Kepler supports both GUI-based and batch mode execution. Our architecture supports all three execution modes of Hadoop[5], i.e., local (standalone), pseudo-distributed, and fully-distributed. Users can easily switch execution modes by configuration.

4. CASE STUDY

In this section, we demonstrate the capability of our architecture via the commonly used word count problem [1], which counts the number of occurrences of each word in a large collection of documents. We show how to express word count with our architecture and discuss its execution performance.

4.1 Word Count Workflow in Kepler

The word count workflow in Kepler is presented in Figure 4, which consists of three parts: the top-level workflow, and the Map and Reduce sub-workflows in the MapReduce actor.

The top-level workflow shows the overall logic. We customize the general MapReduce actor for word count, calling it *MapReduce4WordCount*. The local file system paths for the MapReduce input and output data are specified and sent to the MapReduce4WordCount actor through its input ports. The output path is also connected to the trigger port of the MapReduce4WordCount actor to start its execution. The output port of the MapReduce4WordCount actor is connected to the trig-

[5] http://hadoop.apache.org/common/docs/current/quickstart.html

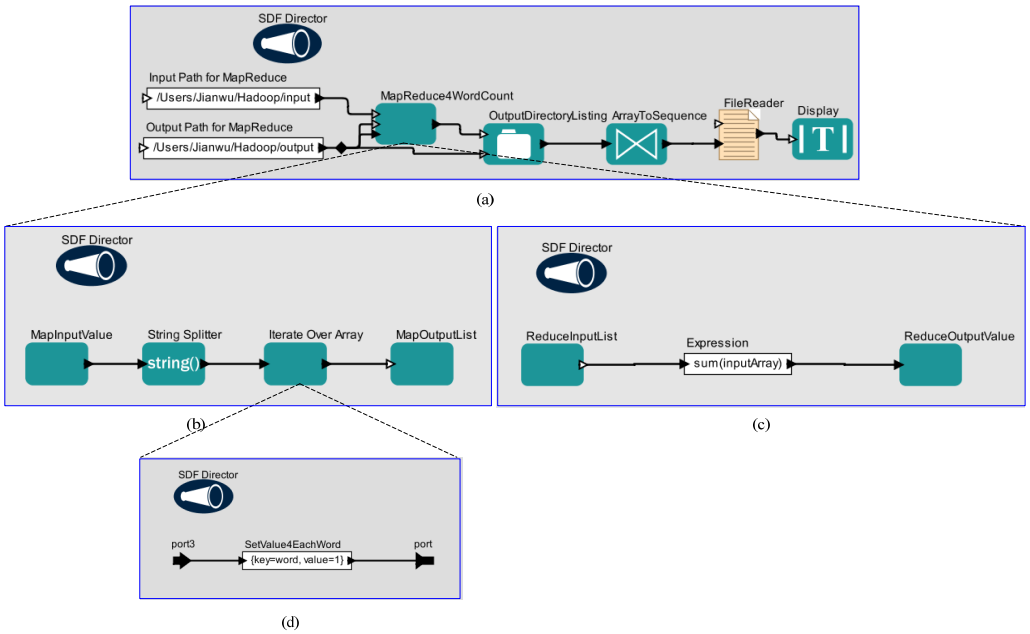


Figure 4: (a) Word count workflow in Kepler. (b) Map sub-workflow. (c) Reduce sub-workflow. (d) Sub-workflow in the IterateOverArray actor.

ger port of the *OutputDirectoryListing actor*; when the execution of the MapReduce4WordCount actor completes, the files in the output path are listed, read and shown by the downstream actors. We choose the SDF director here since the actors in the top-level will be executed sequentially. The Map sub-workflow shows the *Map* logic for word count that will be executed in Hadoop. Each Map sub-workflow instance receives a different portion of the input text distributed by Hadoop; the MapInputValue actor sends out the input text as a token containing a string. The *StringSplitter* actor then separates the string into individual words, and the *IterateOverArray* actor generates key-value pairs for each word (the key is the word and the value is one). The key-values pairs are transferred back to Hadoop for Reduce via the MapOutputList actor.

The Reduce sub-workflow contains the *Reduce* logic. The *ReduceInputList* actor produces an array containing the key-value pairs generated in Map, where the values of each element need to be accumulated. The *Expression* actor sums the values of all elements in the array, which will be output of this sub-workflow.

This use case demonstrates the simplicity of designing MapReduce workflows in our architecture. Based on the extensive actor library in Kepler, users can easily compose MapReduce workflows that solve their domain-specific problems. Furthermore, customized MapReduce actors, like the MapReduce4WordCount actor, can be easily reused and shared locally by choosing the 'Save in Library' item of its right click menu or shared publicly with the 'Upload to Repository' item. Moreover, users can utilize Hadoop transparently in our architecture. For instance, paths in HDFS do not need to be specified in this workflow, and by default, data will be automatically transferred to HDFS before MapReduce execution, and transferred back to the local file system after execution for downstream processing.

4.2 Execution Experiments

In this sub-section, we describe the execution performance of our architecture for the word count use case on a commodity cluster. The cluster has 20 nodes, each with 3.2 GB of memory and two 2.4 GHz CPUs. For the input datasets, we use protein data bank files in the biochemistry domain, which include 1391 files, each with 3758 lines. The total input dataset size is about 348 MB. The test is uses Hadoop 0.20.0, each slave node is configured to have two concurrent Map tasks and one Reduce task, and the heap space is set as 1 GB.

In the first experiment, we measured the scalability of the word count workflow. We also tested the sample implementation in Java for word count case from Hadoop project to see the performance difference. In order to measure execution time only, input data for all these experiments are transferred to HDFS beforehand, and Kepler workflows are executed in batch mode without counting execution time of downstream actors after the MapReduce4WordCount actor.

Figure 5 shows the execution times for this experiment. There were around 1410 map tasks and 18 reduce tasks totally[6]. As the number of slave nodes increases, the execution times of both the Java implementation and our Kepler workflow decreases. Comparing the execution times with and without Kepler, we can see that it takes about four to six times longer using our architecture. Our investigation shows that this overhead is primarily due to Kepler engine initialization and Map/Reduce sub-workflow parsing. The whole execution for the 348 MB input data invoked the Map/Reduce function about 20 million times, so the Map/Reduce sub-workflows were also executed for the same

[6] The Map and Reduce task numbers varies a little for different executions because some duplicate tasks are generated by Hadoop automatically for fault tolerance.

number of times. Although the overhead for each Map or Reduce sub-workflow instance only takes about 10 milliseconds, the accumulative time is substantial compared to the execution time of word count implemented in Java, which is about 0.3 milliseconds for each Map function invocation, and 0.03 milliseconds for each Reduce function invocation.

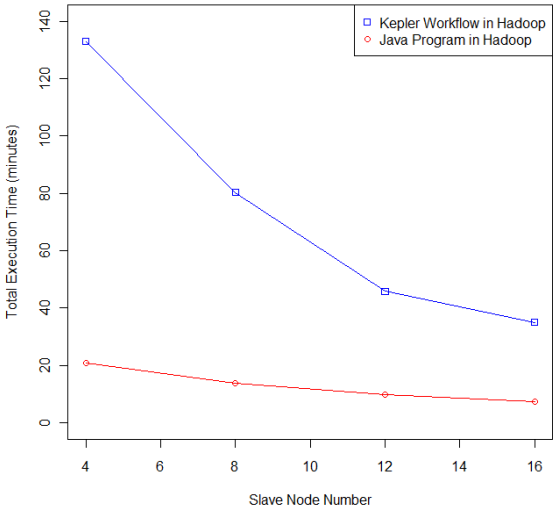


Figure 5: Experiment 1: execution of word count case.

However, the characteristics of the common scientific problems are not fully modeled in this experiment. First, each Map function invocation only processed one line from its input data, whereas a more meaningful data unit for scientific problems is usually much larger. Second, the execution time of the Map and Reduce functions took less than 0.5 milliseconds, whereas a scientific computation unit usually lasts much longer.

To simulate with larger data units in the Map function, we increased the data size from 1 to 50, 100, 500, and 1000 lines, and whole files (3758 lines) for each Map function invocation. The second experiment used the same input data of the first experiment and ran on 16 slave nodes. As shown in Figure 6, both executing with and without Kepler is faster if more data is processed in each Map function invocation, but it becomes slower if too much data is processed. We believe this speedup is due to the smaller overhead percentage along with more processing time for each Map function invocation. The slowdown resulting from sending larger data sizes is due to the decreased concurrency along with decreased the number of total Map function invocations.

To simulate with longer computation time in Map and Reduce functions, we added a 0, 250, 500, 750, and 1000 milliseconds sleep for each Map function invocation and 10 milliseconds sleep for Reduce[7] in the third experiment. Each Map function invocation processed 50 line input. The third experiment used the same input data and cluster configuration in the second experiment. The results, shown in Figure 7, demonstrate that the overhead percentage with Kepler is approximately 10-15% if execution time and data unit size for each Map/Reduce function is not trivially small. Moreover, the overhead percentage decreases as the individual Map/Reduce execution time increases (from 16.13% for 10 milliseconds sleep down to 10.97% for 1010 milliseconds sleep).

[7] The Reduce sleep time is configured to be much smaller than the Map sleep time because the Map phase is usually much longer than Reduce in most map/reduce applications.

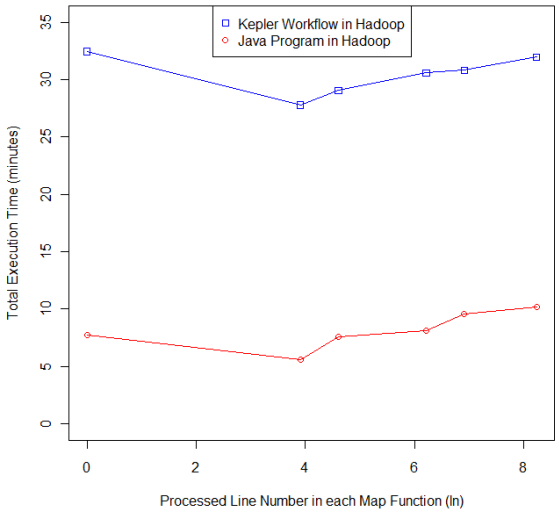


Figure 6: Experiment 2: execution with increased data size in Map.

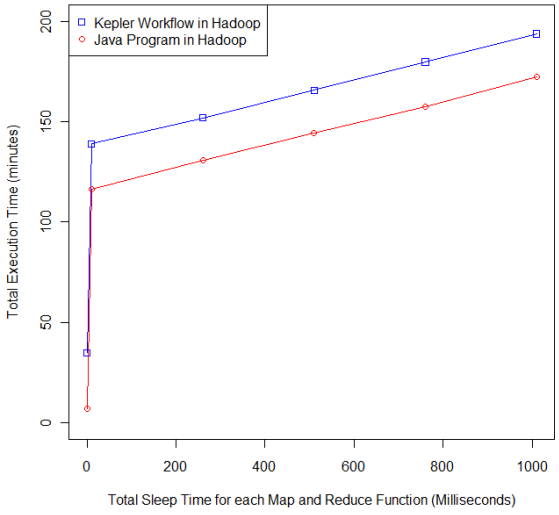


Figure 7: Experiment 3: execution with increased execution time in Map and Reduce.

We note that the above experiments only demonstrate the execution time characteristics of applications using our architecture. Another important aspect is application construction time. Our experience shows that using Kepler can largely reduce

the construction time by its intuitive GUI, especially for domain scientists who are not familiar with programming languages.

5. RELATED WORK

Due to the powerful programming model and good execution performance of MapReduce and Hadoop, there has been an increased effort to enable or utilize them in scientific workflow systems. These projects can be broadly classified into two categories.

Several workflow systems have begun to support MapReduce or similar programming models, such as *map* and *reduce* constructs in VIEW [11], *map*, *foldr* and *foldl* constructs in Martlet [12], *IterateOverArray* actor in Kepler/Ptolemy II [8], and implicit iteration in Taverna [13]. The Master-Salve architecture in Kepler [10] and Service Distribution in Triana [14] can distribute data to multiple remote engines and run them in parallel. Scientific Workflow systems like Kepler [8], Pegasus [15], Swift [16], ASKALON [17] also support parallel job execution. However, to the best of our knowledge, none of the existing scientific workflow systems have been integrated with Hadoop. This means parallel execution on partitioned data is either not supported, or needs to be fully handled by the workflow systems themselves. By following the rule of "separation of concerns" in our architecture, Kepler provides MapReduce workflow composition and management in a graphical user interface while Hadoop takes care of MapReduce execution in distributed environments, so that characteristics of Hadoop such as automatic data partition and distribution, load balance and failure recovery can be naturally embraced in our architecture.

There are studies to integrate Hadoop with workflow systems, but these efforts usually focus on a certain data organization or domain-specific problems, which restricts their generality. [18] proposes and compares different strategies for compiling XML data processing pipelines to a set of MapReduce tasks (implemented within Hadoop) for efficient execution. MRGIS [7] discusses how to use Hadoop for better performance in the geoinformatics domain, only script-based geoinformatics applications can be scheduled and submitted to Hadoop. Our architecture provides a general MapReduce actor where Map/Reduce functions can be easily expressed by sub-workflows in the Kepler GUI and the sub-workflows will be executed in Hadoop. There is no restriction on the data structure in our architecture, and complex data structures can be supported by extending the Hadoop input/output format interfaces. Further, by customizing this actor, domain-specific MapReduce actors can be easily created, reused and shared.

6. CONCLUSION AND FUTURE WORK

Domain scientists greatly benefit from enhanced capability and usability of scientific workflow systems. By leveraging the workflow composition and management capabilities of Kepler, and the execution characteristics of Hadoop, we propose a general and easy-to-use architecture to facilitate data-intensive applications in scientific workflow systems. Scientists can easily create MapReduce sub-workflows, connect them with other tasks using Kepler, and execute them efficiently and transparently via the Hadoop infrastructure. Parallel execution performance can be achieved without bringing its complexity to users. The word count example validates the feasibility of our architecture, which facilitates MapReduce application construction and management with about 10% execution overhead for non-trivial applications.

With the promising results so far, we will enhance our architecture in the future in several directions. The current prototype of our architecture will be refactored to enhance its capability, performance, and robustness. We are applying our architecture to concrete domain-specific scientific problems of ecology and bioinformatics in our ongoing projects. We are also working on enabling distributed provenance recording and querying in our architecture through Bigtable [19] and its Hadoop project HTable [20].

7. ACKNOWLEDGMENTS

The authors would like to thank the rest of the Kepler team for their collaboration, and Daniel Zinn for his feedback. This work was supported by NSF SDCI Award OCI-0722079 for Kepler/CORE, NSF CEO:P Award No. DBI 0619060 for REAP, DOE SciDac Award No. DE-FC02-07ER25811 for SDM Center, and UCGRID Project.

8. REFERENCES

[1] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 137-150. USENIX Association, 2004.

[2] Apache Hadoop Project. http://hadoop.apache.org/core/.

[3] I. Gorton, P. Greenfield, A. Szalay, and R. Williams. Data-Intensive Computing in the 21st Century. In *Computer*, 41(4):30-32, Apr. 2008, doi:10.1109/MC.2008.122.

[4] R. E. Bryant. Data-intensive Supercomputing: The Case for DISC. *Technical Report CMU-CS-07-128*, Carnegie Mellon University, 2007.

[5] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for Data Intensive Scientific Analyses. In *Proceedings of the 4th IEEE International Conference on eScience (e Science 2008)*, pages 277-284, 2008.

[6] M. C. Schatz. Cloudburst: Highly Sensitive Read Mapping with MapReduce. In *Bioinformatics* 2009 25(11):1363-1369. Apr. 2009.

[7] Q. Chen, L. Wang, and Z. Shang. MRGIS: A MapReduce-Enabled High Performance Workflow System for GIS. In *Proceedings of Workshop SWBES08: Challenging Issues in Workflow Applications*, *the 4th IEEE International Conference on eScience (e Science 2008)*, pages 646-651, 2008.

[8] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. B. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. In *Concurrency and Computation: Practice and Experience*, 18(10):1039-1065, 2006.

[9] P. R. Hosseini. Pattern Formation and Individual-Based Models: The Importance of Understanding Individual-Based Movement. In *Ecological Modeling*. 194(4): 357-371. doi:10.1016/j.ecolmodel.2005.10.041. 2006.

[10] J. Wang, I. Altintas, P. R. Hosseini, D. Barseghian, D. Crawl, C. Berkley, and M. B. Jones. Accelerating Parameter Sweep Workflows by Utilizing Ad-hoc Network Computing Resources: an Ecological Example. In *Proceedings of IEEE 2009 Third International Workshop on Scientific Workflows*

(SWF 2009), *2009 Congress on Services (Services 2009)*, pages 267-274, 2009.

[11] X. Fei, S. Lu, and C. Lin. A MapReduce-Enabled Scientific Workflow Composition Framework. In *Proceedings of 2009 IEEE International Conference on Web Services (ICWS 2009)*, pages 663-670, 2009.

[12] D. J. Goodman. Introduction and Evaluation of Martlet: a Scientific Workflow Language for Abstracted Parallelisation. In *Proceedings of the 16th International Conference on World Wide Web (WWW 2007)*, pages 983-992, 2007.

[13] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. In *Bioinformatics*, 20(17), pages 3045-3054, Oxford University Press, London, UK, 2004.

[14] I. Taylor, M. Shields, I. Wang, and O. Rana. Triana Applications within Grid Computing and Peer to Peer Environments. In *Journal of Grid Computing*, 1(2):199-217. Kluwer Academic Press, 2003.

[15] E. Deelman, G. Mehta, G. Singh, M. Su, and K. Vahi. Pegasus: Mapping Large-Scale Workflows to Distributed Resources. In I. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 376-394. Springer, New York, Secaucus, NJ, USA, 2007.

[16] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, Reliable, Loosely Coupled Parallel Computation. In *Proceedings of 2007 IEEE Congress on Services (Services 2007)*, pages 199-206, 2007.

[17] J. Qin and T. Fahringer. Advanced data flow support for scientific grid workflow applications. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC2007),* Article No. 42. 2007.

[18] D. Zinn, S. Bowers, S. Köhler, and B. Ludäscher. Parallelizing XML Processing Pipelines via MapReduce. In Special issue on Scientific Workflows *Journal of Computer and System Sciences*, 2010. Accepted for publication.

[19] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 205-218. USENIX Association, 2006.

[20] Apache HBase Project: http://hadoop.apache.org/hbase/.