International Conference on Computational Science, ICCS 2012

A Framework for Distributed Data-Parallel Execution in the Kepler Scientific Workflow System

Jianwu Wang, Daniel Crawl, Ilkay Altintas*

San Diego Supercomputer Center, UCSD, 9500 Gilman Drive, MC 0505, La Jolla, CA 92093, U.S.A.

Abstract

Distributed Data-Parallel (DDP) patterns such as MapReduce have become increasingly popular as solutions to facilitate data-intensive applications, resulting in a number of systems supporting DDP workflows. Yet, applications or workflows built using these patterns are usually tightly-coupled with the underlying DDP execution engine they select. We present a framework for distributed data-parallel execution in the Kepler scientific workflow system that enables users to easily switch between different DDP execution engines. We describe a set of DDP actors based on DDP patterns and directors for DDP workflow executions within the presented framework. We demonstrate how DDP workflows can be easily composed in the Kepler graphic user interface through the reuse of these DDP actors and directors and how the generated DDP workflows can be executed in different distributed environments. Via a bioinformatics usecase, we discuss the usability of the proposed framework and validate its execution scalability.

Keywords: Scientific workflows, distributed data-parallel patterns, data-intensive, bioinformatics

1. Introduction

Rapid advances in data observation, collection, and analysis technologies have led to an enormous growth in the amount of scientific data. Examples of new technologies and platforms include satellite and ground-based sensor observatories collecting data in real-time, simulations running on petascale supercomputers, and web-scale data-mining algorithms. Next-generation DNA sequencers [1] used by bioinformaticians generate large amounts of sequence data; for example, the Illumina HiSeq 2500 can produce six billion paired-end reads per run (600 Gb)[1].

Most existing domain-specific codes do not scale at this magnitude of data generation, so scientists must rely on distributed and parallel processing methodologies for comprehensive data analysis. A common technique when working with large-scale data is to split input data into smaller jobs that are executed in Cluster, Grid, or Cloud environments, and then merge the results. New computational techniques and efficient execution mechanisms for this data-intensive workload are needed. Technologies such as data-intensive computing [2] and scientific workflows [3] have the potential to enable rapid data analysis for many scientific problems.

---

*Corresponding author. Tel.: +1-858-822-5453; fax: +1-858-822-3693

Email address: {jianwu, crawl, altintas}@sdsc.edu (Jianwu Wang, Daniel Crawl, Ilkay Altintas)
[1]Illumina HiSeq 2500: http://www.illumina.com/systems/hiseq_systems.ilmn, 2012.

1.1. Distributed Data-Parallel Patterns and Execution Engines

Many distributed data-parallel patterns identified recently provide opportunities to facilitate data-intensive applications/workflows, which can execute in parallel with split data on distributed computing nodes. Examples of these patterns include MapReduce [4], All-Pairs [5], Sector/Sphere [6] and PACT [7]. The advantages of these patterns include: *(i)* support data distribution and parallel data processing with distributed data on multiple nodes/cores; *(ii)* provide a higher-level programming model to facilitate user program parallelization; *(iii)* follow a "moving computation to data" principle that reduces data movement overheads; *(iv)* have good scalability and performance acceleration when executing on distributed resources; *(v)* support run-time features such as fault-tolerance; *(vi)* simplify the difficulty for parallel programming in comparison to traditional parallel programming interfaces such MPI [8] and OpenMP [9]. If proper DDP patterns can be applied, existing single node programs could be executed in parallel without modification. An example will be shown later in Section 5.1.

There are an increasing number of execution engines that implement the above distributed data-parallel patterns. MapReduce is a representative example that has different DDP execution engine implementations. A popular MapReduce execution engine is Hadoop[2]. Other MapReduce implementation projects include Cloud MapReduce[3], Phoenix[4] and MapReduce-MPI[5]. The Stratosphere system[6] also supports MapReduce as part of their PACT programming model.

1.2. Kepler Scientific Workflow System

The Kepler[7] scientific workflow system is an open-source, cross-project collaboration to serve scientists from different disciplines [10, 11]. Kepler inherits an actor-oriented modeling paradigm from Ptolemy II[8] [12], and extends it for the design and execution of scientific workflows. Since its initiation in 2003, Kepler has been used in a wide variety of projects to manage, process, and analyze scientific data.

Kepler provides a graphical user interface (GUI) for designing scientific workflows, which are a structured set of steps or tasks linked together that implement a computational solution to a scientific problem. In Kepler, *Actors* provide implementations of specific tasks and can be linked together via input and output *Ports*. Data is encapsulated in messages or *Tokens*, and transferred between actors through ports. Actor execution is governed by *Model of Computations* (MoCs), which are realized as *Directors* [13].

The MapReduce actor [14] provides an interface for building workflows that use Hadoop's MapReduce. Since Map and Reduce are two separate data-parallel patterns, they are treated as two independent sub-workflows in the MapReduce actor. The Kepler GUI displays Map and Reduce sub-workflows inside of the MapReduce actor as separate components. The actor can be placed in a larger workflow to connect the MapReduce functionality with the rest of the workflow. Each MapReduce actor will be run as a standalone Hadoop job, which calls the Kepler engine to execute the Map and Reduce sub-workflows.

1.3. Challenges and Contributions

The DDP execution engines described in Section 1.1 employ similar data-parallel patterns, but have different capabilities and characteristics, *e.g.*, fault-tolerance, performance, etc. As a result, applications built on one system may not run efficiently or correctly on another system, forcing users to decide which implementation to use before they build their DDP applications. Additionally, users might need to switch DDP systems for better performance or due to the software stack availability on different computational resources. With a goal to provide more flexible DDP programming and execution capabilities, we present a framework for distributed data-parallel execution in the Kepler scientific workflow system where a workflow can be executed on different underlying DDP execution engines after a few quick changes in its configuration. Using a separation of concerns principle, the design of actor-oriented

---

[2]Hadoop Project: http://hadoop.apache.org, 2012.

[3]Cloud MapReduce Project: http://code.google.com/p/cloudmapreduce/, 2012

[4]Phoenix System: http://mapreduce.stanford.edu/, 2012.

[5]MapReduce-MPI: http://www.sandia.gov/~sjplimp/mapreduce.html, 2012

[6]Stratosphere Project: http://www.stratosphere.eu/, 2012.

[7]Kepler website: http://kepler-project.org/, 2012.

[8]Ptolemy II website: http://ptolemy.berkeley.edu/ptolemyII/, 2012.

workflows in Kepler is separated from the computation engines that execute the workflow. We use this unique feature of Kepler (inherited from Ptolemy II) to achieve flexible DDP workflow execution that can utilize different DDP engines by simply switching the director within a DDP sub-workflow.

The contributions of this paper are: *(i)* an architecture for the DDP framework; *(ii)* implementations of DDP actors based on DDP patterns and directors for the execution of DDP workflows; *(iii)* discussion on the usage of the DDP framework and its usability; and *(iv)* usecase and preliminary experimental results.

Outline. The rest of this paper is organized as follows. In Section 2, we discuss related work. Section 3 describes the components in the architecture of our framework. Two key components in the architecture, DDP actors and directors, are explained in Section 4. In Section 5, we provide one example usecase and show its experimental results in a Cluster environment. Section 6 explains how to use the Kepler DDP framework and describes its usability. In Section 7, we discuss our conclusions and plans for future work.

2. Related Work

Many scientific applications have been built on the Hadoop execution engine. Applications such as CloudBurst [15] and Crossbow [16] rebuild traditional single-processor bioinformatics tools using the MapReduce parallel pattern. These applications demonstrate the capability of data-parallel patterns, *e.g.*, Map and Reduce, to parallelize execution of traditional bioinformatics tools on multiple compute nodes.

Over the last couple of years, several efforts have built high-level abstractions on the Hadoop execution engine including our own MapReduce actor in Kepler [14]. Pig Latin is a programming language for expressing data processing tasks [17]. The Pig platform compiles Pig Latin programs into MapReduce jobs for Hadoop. The Nova workflow system is designed for batched, incremental processing of large datasets [18], and is built on Pig and Hadoop. In a Nova workflow, tasks read and write data to the Hadoop Distributed File System (HDFS) using one of several patterns, *e.g.*, non-incremental, stateful incremental, etc. Nova workflows are translated into Pig Latin programs, which are compiled and executed as Hadoop jobs. The Oozie project[9] is a workflow system to manage Hadoop jobs. The Cascading project[10] supports data flow paradigm using DDP patterns such as Each, GroupBy, and CoGroup, and transforms the user generated data flow logic into MapReduce jobs for Hadoop.

Several execution engines support DDP patterns besides Hadoop. In Stratosphere, DDP patterns are expressed using the Parallelization Contract (PACTs) programming model [7], and the PACT compiler converts PACT programs into jobs for the Nephele execution engine [19]. DryadLINQ [20] is a high-level programming language consisting of .NET constructs for operating on datasets and runs on Dryad [21], a distributed data flow execution engine. Other available DDP execution engines include Cloud MapReduce, Phoenix and MapReduce-MPI.

However, to the best of our knowledge, none of the above applications or systems support workflow execution on more than one DDP execution engine. By providing a clean separation between workflow composition and workflow execution, our framework can easily run the same workflow on different computational platforms. In addition, the above systems can be hard to use for users without much programming expertise. Through the Kepler GUI and actor/workflow repository management, the framework in this paper provides an easy way for workflow composition, sharing, and reuse.

3. General Architecture

In this section we describe the architecture for the Kepler distributed data-parallel framework. Figure 1 shows the major components organized in three layers explained in this section: the modular Kepler scientific workflow system including DDP actors and directors, DDP execution engines such as Hadoop and Stratosphere, and various computational environments.

---

[9]Oozie Project: http://incubator.apache.org/oozie/index.html, 2012.
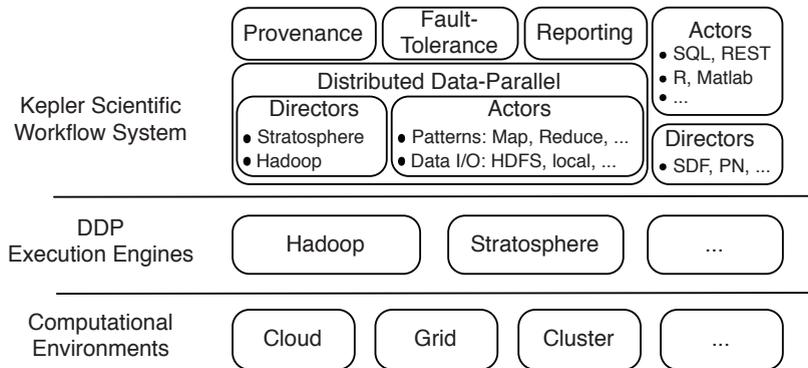[10]Cascading Project: http://www.cascading.org/, 2012.

Figure 1: Architecture of distributed data-parallel framework in Kepler.

3.1. Kepler Modules and DDP Extensions

The Kepler scientific workflow system is organized as a set of extensions or *modules*. The *Actors* module contains a large library of actors that can perform, *e.g.*, data I/O, analysis, visualization tasks, etc. The *Directors* module provides the implementation of several models of computation such as Synchronous Data Flow (SDF) [22] and Process Networks (PN) [23]. The *Provenance* module provides a framework to capture and query workflow provenance including the workflow structure, *e.g.*, actors, parameters, etc., and the data transferred between actors during workflow execution [24]. The *Reporting* module uses provenance information collected during workflow execution to generate a report displaying the workflow results. The *Fault-Tolerance* module contains mechanisms for detecting and handling errors during workflow execution. This module includes a special *Contingency* actor, which allows workflow developers to implement alternative workflows that execute during error conditions [25].

In this paper, we describe a new module for Kepler that supports DDP execution. This module includes a set of DDP actors and directors and will be discussed Section 4.

3.2. DDP Execution Engines

As explained in Sections 1.1 and 2, there are many different DDP execution engines available. In this paper, we focus on two DDP execution engines: Hadoop and Stratosphere.

Hadoop is an open-source implementation of MapReduce [4]. It uses HDFS to provide data distribution and redundancy on Hadoop nodes. One Hadoop node, called *Master*, dispatches tasks and executes them in parallel on the other Hadoop nodes, *i.e.*, *Slaves*. The Hadoop Master node monitors task execution and enables fault-tolerant execution by re-executing failed tasks on other Slave nodes.

The Stratosphere system is comprised of two main parts: the Parallelization Contract (PACT) programming model [7] and Nephele execution engine [19]. A PACT contains three components: *(i)* an input contract which specifies the data-parallel pattern, *e.g.*, Map, Reduce, etc.; *(ii)* a user-defined first-order function that provides the processing steps; and *(iii)* an optional output contract that aids optimizing overall job execution. The PACT compiler converts PACT programs into Nephele graphs, where each node is a processing task and edges represent data movement. Nephele has the same node role assignment using Master and Slave like Hadoop. The Nephele execution engine can dynamically allocate or deallocate resources as needed, and run with HDFS to achieve data distribution and locality, but may also transfer data between tasks via shared memory or direct network connections.

Through HDFS, data will be partitioned into blocks and distributed on compute nodes. Both Hadoop and Stratosphere support customized data partitioning for reading and writing. Tasks on each Hadoop or Stratosphere Slave node will try to run against the data blocks on HDFS that is accessible locally.

3.3. Computational Environments

Many DDP execution engines, *e.g.* Hadoop and Stratosphere, can run in different computational environments including Clouds, Grids, Clusters, and even hybrid ones. This portability of DDP execution engines enables our DDP

framework to be portable from one computational environment to another. For instance, the framework can be used first in a local Cluster and, later, it can be easily migrated to a Cloud environment for better scalability.

4. DDP Actors and Directors

4.1. DDP Actors

The Kepler DDP framework includes a set of specialized actors that provide an interface to data-parallel patterns. Each DDP actor corresponds to a particular data-parallel pattern, and there are actors for Map, Reduce, Cross, CoGroup, and Match. The semantics of these data-parallel patterns are the same with the input contracts in the PACT programming model [7]. Similar to other actors, the DDP actors can be linked together to form a chain of tasks, and can be nested hierarchically as part of larger workflows. Unlike our existing MapReduce actor, where the Map and Reduce patterns are bundled together, each DDP actor corresponds to a single pattern thereby allowing greater flexibility to express workflow logic. For example, one or more DDP actors can be dependent on one upstream Map actor.

A DDP actor provides two important pieces of information: the data-parallel pattern, and the tasks to perform once the pattern has been applied. The type of data-parallel pattern specifies how the data is grouped among computational resources. For example, Reduce combines all keys with the same value into a single group, but Map performs no grouping and processes each key independently.

Once the data is grouped, a set of processing steps are applied to the data. DDP actors provide two mechanisms for users to specify these steps: choose a predefined function or create a sub-workflow. In the former case, users choose from a library of predefined functions to process or analyze the data. Developers can write their own functions in Java and add these to the library. Additionally, DDP actors can use a sub-workflow to specify the data processing tasks. In this case, the user builds a sub-workflow in Kepler using Kepler actors and directors.

The DDP framework also includes I/O actors for reading and writing data between the data-parallel patterns and the underlying storage system. The *FileDataSource* actor specifies the location of the input data in the storage system as well as how to partition the data. Similarly, the *FileDataSink* actor specifies the data output location and how to join the data. The partitioning and joining methods depend on the format of the data and are application-specific. The DDP framework includes a library of common methods from which the user can choose. New methods can be also imported. The FileDataSource and FileDataSink actors currently support HDFS and the local file system.

4.2. DDP Directors

In order to run workflows composed of DDP actors, two DDP directors have been implemented. The *StratosphereDirector* and *HadoopDirector* execute workflows on the Stratosphere and Hadoop execution engines, respectively. Concretely, the tasks running on each Stratosphere and Hadoop Slave node will first assign input data to the predefined function or sub-workflow defined in each DDP actor, and then call Kepler execution engine to execute it. We are designing a third director that automatically chooses the underlying DDP execution engine based on several factors including the computational environment and dataset sizes.

4.2.1. Stratosphere Director

The *StratosphereDirector* converts DDP workflows into the PACT programming model, which are then compiled into Nephele jobs. To convert a workflow into the PACT programming model, the *StratosphereDirector* first translates each DDP actor to the corresponding PACT input contract, *e.g.*, the Map actor is translated to the Map input contract. Next, the director uses the processing steps in the DDP actor to specify the first-order function for the PACT contract. If the steps are from the library of predefined functions, then this function can be directly used as the first-order function for the PACT contract. Otherwise if the steps are represented as a sub-workflow, the *StratosphereDirector* configures first-order function to run Kepler. In this case, when the PACT contract executes in Nephele, it calls the Kepler execution engine to load and execute the sub-workflow specified in the DDP actor.

Since Kepler inherits Ptolemy II's data type system, actors' inputs and outputs have a specific data types, *e.g.*, integer, float and string. Two actors can be connected only if their input and output data types are compatible. Similarly, a PACT contract also has data types for inputs and outputs, and these must be configured before Nephele can execute the task. When the *StratosphereDirector* translates a DDP actor into a PACT contract, the PACT data types

are configured to be the corresponding Kepler data types used by the DDP actor. Similar primitive and composite data types exist in Kepler and PACT, and mapping between them is straightforward.

If a Kepler sub-workflow is used as the first-order function for a PACT contract, the Kepler execution engine executes the sub-workflow as part of the Nephele task. When the task executes, Nephele writes input data to Kepler, and reads any output data from Kepler.

4.2.2. Hadoop Director

The first challenge for the *HadoopDirector* is to build Hadoop jobs based on general DDP actors in Section 4.1. Unlike Stratosphere where each DDP actor naturally corresponds to a PACT input contract, each Hadoop job includes one Map task and an optional Reduce task. Therefore the logic of DDP actors in a Kepler workflow needs to be transformed into that of Hadoop jobs.

The second challenge for the *HadoopDirector* is to manage execution dependencies of DDP actors in a workflow, which includes actor execution order and data transfer between actors. Hadoop itself only runs one Hadoop job for each execution, whereas Stratosphere can process a whole DDP workflow for each execution. Dependencies between Hadoop jobs in a workflow need to be managed by the *HadoopDirector*.

The *HadoopDirector* could be built either from scratch or by extending existing directors in Kepler. Currently, we built our *HadoopDirector* by reusing existing directors and the MapReduce actor [14] in Kepler. The *HadoopDirector* 1) first transforms a workflow with DDP actors into a workflow with MapReduce actors; 2) then uses an existing Kepler director (*e.g.*, SDF) to manage workflow execution. This approach enables us to reuse the implementation logic of the existing MapReduce actor to embed Map and Reduce sub-workflows into Hadoop jobs. The usage of existing Kepler directors provides us with the reuse of their capabilities including logic control and intermediate data buffering. For data type transformation between Kepler and Hadoop, we use a similar approach as we did in the *StratosphereDirector*.

Based on this reuse approach, the focus of the *HadoopDirector* is to transform a workflow with DDP actors into a workflow with MapReduce actors. At this stage, only Map and Reduce patterns are supported in the *HadoopDirector*. For other patterns, *e.g.*, Cross, which needs more than one data input, we plan to support them by using Hadoop features like DistributedCache[11]. The main workflow transformation logic is as follows. Since DDP workflows have to start with FileDataSource actors to read partitioned data, the director first finds all FileDataSource actors, and then traverses the downstream actors from each FileDataSource actor until the end of branch, namely FileDataSink actor. During the traversals, Map and Reduce actors are merged into MapReduce actors if one Map actor is followed by only one Reduce actor. If no merge happens, a Map actor will be transformed into a map-only MapReduce actor without a reduce sub-workflow, and a Reduce actor into a MapReduce actor with a transparent map sub-workflow, which simply sends data received from Map's inputs to its outputs.

We acknowledge that there could be more sophisticated workflow transformation approaches, such as merging multiple connected Maps and Reduces actors into one MapReduce actor. As a part of our future work, we will study whether we can improve the workflow transformation approach to optimize workflow execution performance.

4.2.3. Generic DDP Director

We are currently designing a generic *DDP Director* based on the *HadoopDirector* and the *StratosphereDirector*. The director will first detect the availability of Hadoop or Stratosphere execution engine and use the corresponding director when only one is available. If both are available, the generic director will determine which one is best using factors such as historical execution statistics, data size, and intermediate data storage resources. The generic director will be more user-friendly since it completely hides the underlying execution engines. Further, this generic DDP director can be extensible to support other or newer DDP execution engines such as Cloud MapReduce.

5. Experimental Application

5.1. A DDP Workflow in Bioinformatics

To validate the proposed framework for scientific applications, we built a workflow that runs BLAST, a tool that detects similarities between query sequence data and reference sequence data [26]. Executing BLAST can be very

---

[11]DistributedCache in Hadoop: http://hadoop.apache.org/common/docs/current/mapred_tutorial.html#DistributedCache, 2012.

data-intensive since the query or reference data may have thousands to millions of sequences. To parallelize BLAST execution, a typical way is to partition input data and run the BLAST program in parallel against the data partitions. In the end, the results need to be aggregated. Partitioning could be done for the query data, reference data, or both. We have discussed a DDP BLAST workflow via partitioning the query data in [27]. In this paper, we build a DDP BLAST workflow that partitions the reference data.

(a) Top level

(b) Map

(c) Reduce
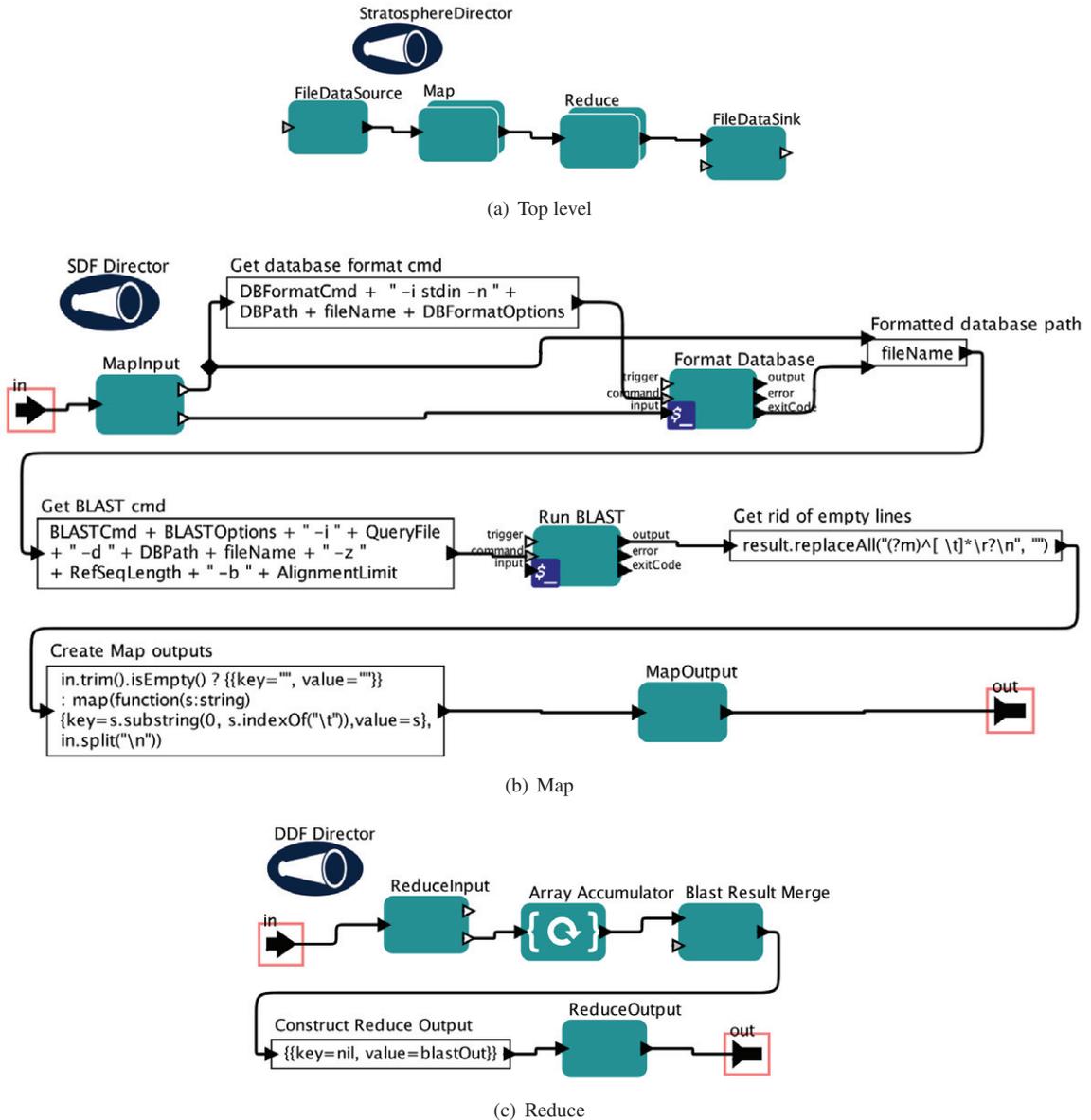
Figure 2: A Distributed Data-Parallel BLAST workflow.

Figure 2(a) shows the overall DDP BLAST workflow. The FileDataSource actor is configured to read reference sequence data in HDFS, and generate key value pairs for the Map actor. We built customized partitioning method to split the reference data on sequence boundaries so that each Map instance will read whole sequences. Both Map and Reduce actors have sub-workflows as shown in Figures 2(b) and 2(c), respectively. In each execution of Map/Reduce sub-workflow, it will read a key value pair from its input actor, and generate key value pairs to its output actor. For each set of reference sequences gotten from its MapInput actor, the Map sub-workflow first converts them into a binary for-

mat, and then executes BLAST against the formatted reference data and whole query data. The outputs from BLAST are read by the Reduce sub-workflow, which sorts and merges them. Finally, the FileDataSink actor writes the sorted outputs into a single file. This workflow can be executed with Hadoop by just replacing the *StratosphereDirector* in Figure 2(a) to be the *HadoopDirector* in Kepler GUI.

5.2. Preliminary Experiments

The DDP BLAST workflow was executed in a compute Cluster environment to measure its scalability. The nodes used in these experiments have two dual-core AMD 2GHz CPUs, 8GB of memory, and run CentOS 5.5 Linux. The nodes can access a shared file system via NFS, which store Kepler, the query sequence data, and the BLAST programs. Reference sequence data is staged in HDFS before execution. The tests were done with Hadoop 0.20.2, Stratosphere 0.1.2, Kepler 2.3, and our Kepler DDP module. For all the experiments, both Hadoop and Stratosphere were configured to run four Map and one Reduce instance on each node so that we can utilize all four cores of each node for Map instances. In our experiments, the data sizes of query and reference data file are 56MB and 244MB, respectively. The block size of HDFS is configured to be 16MB so HDFS will automatically split the 244MB reference data into 16 blocks. In these experiments, *HadoopDirector* is implemented by extending existing SDF director in Kepler.
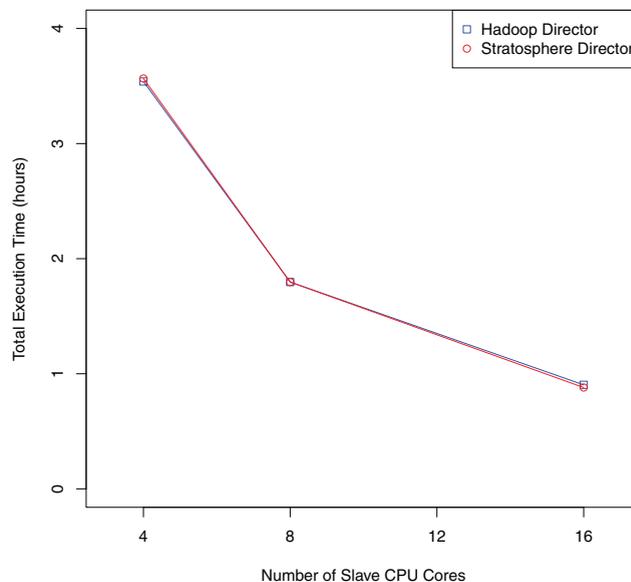

Figure 3: DDP BLAST workflow execution.

Figure 3 shows execution times for the BLAST workflow using different numbers of Slave CPU cores. For both Stratosphere and Hadoop directors, the workflow executions show good scalability and acceleration. The performances using the Stratosphere and Hadoop execution engine are almost the same. In the future we will run more complex DDP workflows with larger-scale datasets on bigger computing environments.

6. Usage of DDP actors and directors

In this section we analyze how a workflow developer can use the presented DDP actors and directors, which is illustrated in Figure 4. The workflow developer first searches the Kepler actor library for DDP actors that match the requirements of the DDP workflow application he/she is building (*Step 1*). The library contains both generic and

domain-specific DDP actors as described in Sections 4.1 and 5.1, respectively. Proper domain-specific DDP actors can be used directly if they exist, *e.g.*, DDP BLAST (*Step 2a*). Otherwise, the workflow developer can build his/her own by creating a sub-workflow using a generic DDP actor (*Step 2b*). In this case, he/she needs to understand the data-parallel patterns and how his/her scientific computational problems can be parallelized. By reusing existing or building new domain-specific DDP actors, the workflow developer can connect them based on their dependencies and select the proper DDP director, which results in an executable domain-specific workflow (*Step 3*). Once the workflow is constructed, the workflow developer can configure actor parameters and specify the locations for input and output data. The workflow can be then executed through the Kepler GUI or on the command-line (*Step 4a*). Additionally, the workflow developer can add their workflow as a sub-workflow in a larger workflow (*Step 4b*), or save it in the actor library so that it can be reused in the future (*Step 4c*). For example, the workflow in Figure 2 can be saved in actor library as a composite actor that performs parallel BLAST execution.
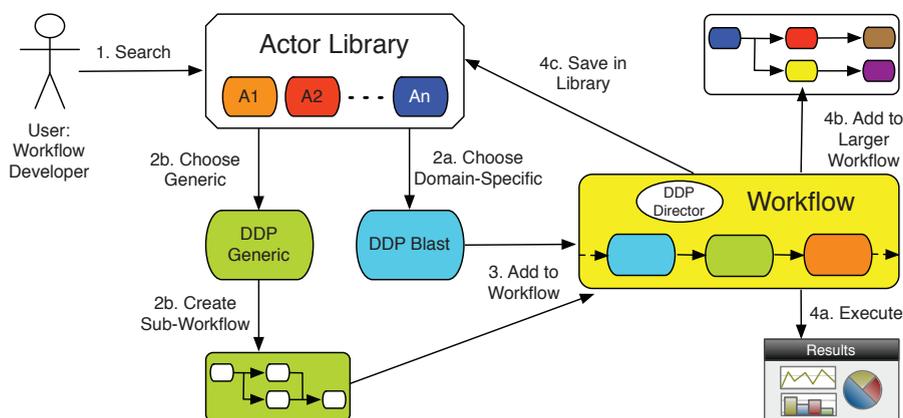

Figure 4: Usage process of DDP actors and directors.

Compared to the related approaches in Section 2, the above usage model is easy-to-use, and has good reusability and adaptation. First, all the above work can be done in the Kepler GUI without writing a single line of external code. Second, reusability can occur at different levels; generic DDP actors, domain-specific DDP actors, and domain-specific DDP workflows can all be shared and reused later. Third, the same domain-specific DDP workflow can be executed with different DDP execution engines by just changing the DDP director.

7. Conclusions and Future Work

Domain scientists greatly benefit from improved capability and usability of scientific workflow systems. By leveraging the workflow composition and management capabilities of Kepler, and the execution characteristics of distributed data-parallel patterns, we propose a general and easy-to-use framework to facilitate data-intensive applications in scientific workflow systems. Scientists can easily create DDP workflows, connect them with other tasks using Kepler, and execute them efficiently and transparently via available DDP execution engines. Parallel execution performance can be realized without bringing its complexity to users. The bioinformatics example validates the feasibility of our framework, which facilitates DDP application construction and management with good scalability in distributed environments.

For future work, we will consolidate our implementation and test it with additional usecases. We plan to support other DDP patterns for the *HadoopDirector* and try to optimize the mapping between DDP actors and Hadoop jobs. We will also work on how to automatically convert a local workflow to a DDP workflow.

8. Acknowledgments

Gordon and Betty Moore Foundation award to Calit2 at UCSD for CAMERA, and an SDSC Triton Research Opportunities grant.

References

[1] J. Shendure, H. Ji, Next-generation DNA sequencing, Nature Biotechnology 26 (10) (2008) 1135–1145.
[2] I. Gorton, P. Greenfield, A. S. Szalay, R. Williams, Data-intensive computing in the 21st century, IEEE Computer 41 (4) (2008) 30–32.
[3] I. J. Taylor, E. Deelman, D. B. Gannon, M. Shields (Eds.), Workflows for e-Science, Springer, 2007.
[4] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113.
[5] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, D. Thain, All-Pairs: An abstraction for data-intensive computing on campus Grids, IEEE Transactions on Parallel and Distributed Systems 21 (2010) 33–46. `doi:http://doi.ieeecomputersociety.org/10.1109/TPDS.2009.49`.
[6] Y. Gu, R. Grossman, Sector and sphere: The design and implementation of a high performance data cloud, Philosophical Transactions of the Royal Society A 367 (1897) (2009) 2429–2445.
[7] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, D. Warneke, Nephele/PACTs: A programming model and execution framework for web-scale analytical processing, in: Proceedings of the 1st ACM symposium on Cloud computing, SoCC '10, ACM, New York, NY, USA, 2010, pp. 119–130. `doi:http://doi.acm.org/10.1145/1807128.1807148`.
[8] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message Passing Interface, 2nd Edition, Scientific And Engineering Computation Series, MIT Press, Cambridge, MA, USA, 1999.
[9] B. Chapman, G. Jost, R. van der Pas, D. Kuck, Using OpenMP: Portable Shared Memory Parallel Programming, The MIT Press, Cambridge, MA, USA, 2007.
[10] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludaescher, S. Mock, Kepler: An extensible system for design and execution of scientific workflows, in: Proceedings of 16th International Conference on Scientific and Statistical Database Management, SSDBM 2004, Santorini Island, Greece, 2004, pp. 423–424.
[11] B. Ludaescher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. A. Lee, J. Tao, Y. Zhao, Scientific workflow management and the Kepler system, Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows 18 (10) (2006) 1039–1065.
[12] E. A. Lee, S. Neuendorffer, M. J. Wirthlin, Actor-oriented design of embedded hardware and software systems, Journal of Circuits, Systems, and Computers 12 (3) (2003) 231–260.
[13] A. Goderis, C. Brooks, I. Altintas, E. Lee, C. Goble, Heterogeneous composition of models of computation, Future Generation Computer Systems 25 (5) (2009) 552–560.
[14] J. Wang, D. Crawl, 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, ACM New York, NY, USA, Portland, Oregon, 2009, pp. 1–8.
[15] M. Schatz, Cloudburst: Highly sensitive read mapping with MapReduce, Bioinformatics 25 (11) (2009) 1363–1369.
[16] B. Langmead, M. C. Schatz, J. Lin, M. Pop, S. L. Salzberg, Searching for SNPs with cloud computing, Genome Biology 10 (134).
[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig latin: a not-so-foreign language for data processing, in: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD '08, ACM, New York, NY, USA, 2008, pp. 1099–1110. `doi:http://doi.acm.org/10.1145/1376616.1376726`.
[18] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, X. Wang, Nova: continuous Pig/Hadoop workflows, in: Proceedings of the 2011 international conference on Management of data, SIGMOD '11, ACM, New York, NY, USA, 2011, pp. 1081–1090. `doi:http://doi.acm.org/10.1145/1989323.1989439`.
[19] D. Warneke, O. Kao, Exploiting dynamic resource allocation for efficient parallel data processing in the Cloud, Parallel and Distributed Systems, IEEE Transactions on 22 (6) (2011) 985 –997. `doi:10.1109/TPDS.2011.65`.
[20] M. Isard, Y. Yu, DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language, in: Proceedings of the 35th SIGMOD international conference on Management of data, SIGMOD '09, ACM, New York, NY, USA, 2009, pp. 987–994.
[21] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, in: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, Vol. 41 of EuroSys '07, ACM, New York, NY, USA, 2007, pp. 59–72. `doi:http://doi.acm.org/10.1145/1272998.1273005`.
[22] E. A. Lee, D. G. Messerschmitt, Synchronous data flow, Proceedings of the IEEE 75 (9) (1987) 1235–1245.
[23] E. A. Lee, T. M. Parks, Dataflow process networks, Proceedings of the IEEE 83 (5) (1995) 773–801.
[24] I. Altintas, O. Barney, E. Jaeger-Frank, Provenance collection support in the Kepler scientific workflow system, in: Proceedings of International Provenance and Annotation Workshop, IPAW 2006, 2006, pp. 118–132.
[25] P. Mouallem, D. Crawl, I. Altintas, M. A. Vouk, U. Yildiz, A fault-tolerance architecture for Kepler-based distributed scientific workflows, in: Proceedings of the 22nd International Conference on Scientific and Statistical Database Management, SSDBM 2010, Springer, Berlin, Heidelberg, 2010, pp. 452–460.
[26] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, D. J. Lipman, Basic Local Alignment Search Tool, Journal of Molecular Biology 215 (3) (1990) 403 – 410. `doi:10.1016/S0022-2836(05)80360-2`.
[27] I. Altintas, J. Wang, D. Crawl, W. Li, Challenges and approaches for distributed workflow-driven analysis of large-scale biological data, in: Proceedings of the Workshop on Data analytics in the Cloud at EDBT/ICDT 2012 Conference, DanaC2012, 2012.