Big Data Applications using Workflows for Data Parallel Computing

Jianwu Wang, Daniel Crawl, Ilkay Altintas, Weizhong Li
University of California, San Diego

Abstract

In the Big Data era, workflow systems need to embrace data parallel computing techniques for efficient data analysis and analytics. We present an easy-to-use, scalable approach to build and execute Big Data applications using actor-oriented modeling in data parallel computing. We use two bioinformatics use cases for next-generation sequencing data analysis to verify the feasibility of our approach.

Keywords

Big data workflow, Actor-oriented programming, Data parallelization, Bioinformatics application

1. Introduction

As the Internet of Things [1] and other data acquisition and generation technologies advance, data being generated is growing at an exponential rate at all scales in many online and scientific platforms. This mostly unstructured and variable data growing and moving between different applications dynamically in vast quantities is often referred to as "Big Data". The amount of potentially valuable information buried in Big Data is of interest to many data science applications ranging from natural sciences to marketing research. In order to analyze and digest such heterogeneous data, challenges for integration and distributed analysis should overcome include: scalable data preparation and analysis techniques; new and distributed programming paradigms; and innovative hardware and software systems that can serve applications based on their needs.

An important aspect of Big Data applications is the variability of technical needs based on applications being developed. These applications typically involving data ingestion, preparation (e.g., extract, transform, and load), integration, analysis, visualization and dissemination are referred to as Data Science Workflows [2]. A data science workflow development is the process of combining data and processes into a configurable, structured set of steps that implement automated computational solutions of an application with capabilities including provenance management, execution management and reporting tools, integration of distributed computation and data management technologies, ability to ingest local and remote scripts, and sensor management and data streaming interfaces. Data science workflows have a set of technology challenges that can potentially employ a number of Big Data tools and middleware. Rapid programmability of applications on a use case basis requires workflow management tools that can interface to and facilitate integration of other tools. New programming techniques are needed for building effective and scalable solutions span across the data science workflows. Flexibility of workflow systems to combine tools and data together makes it an ideal choice for the development of data science applications involving common Big Data programming patterns.

Big Data workflows have been an active research area since the introduction of scientific workflows [2]. After the development and general adoption of MapReduce [3] as a Big Data programming pattern, a number of workflow systems were built or extended to enable programmability of MapReduce applications including Oozie [4], Nova [5], Azkaban[1] and

[1] Azkaban Project: http://azkaban.github.io/azkaban2/, 2014

Digital Object Indentifier 10.1109/MCSE.2014.50 1070-9924/$26.00 2014 IEEE

Cascading[2]. In this article, we present programming extensions on MapReduce and other Big Data programing patterns to the well-adopted Kepler Workflow Environment (https://kepler-project.org/) and the engines built on top of Hadoop[3] and Stratosphere[4] systems to execute workflows including these patterns. The unique part of our approach is: (*i*) its heterogeneous nature in which Big Data programming patterns are placed as part of other workflow tasks; (*ii*) its visual programming approach that does not require scripting of Big Data patterns; (*iii*) its adaptability for executing data parallel applications on different execution engines.

Our tools presented in this article are applicable to Big Data workflows in all domains. By leveraging the workflow composition and management capabilities of Kepler [6], and the execution characteristics of distributed data parallel (DDP) patterns like MapReduce, we provide a general and easy-to-use framework and tool to facilitate Big Data applications in scientific workflow systems. Users 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 achieved transparently by our execution engine without user intervention.

This approach scales to all forms of Big Data including structured, non-structured and semi-structured data. We provide two example use cases on analysis of next-generation sequencing data and other bioinformatics data using community developed bioinformatics tools. The approach is applicable on a range of computing resources including Hadoop clusters, XSEDE and Amazon's EC2. A typical data size for such applications is within gigabyte (GB) to terabyte (TB) range.

2. Data Parallel Computing in Distributed Environments

From algorithmic perspective, several design structures are commonly used in data parallel analysis and analytics applications. To generalize and reuse these design structures in more applications, many DDP patterns have been identified to easily build efficient data parallel applications. Such DDP patterns include *Map*, *Reduce*, *Match*, *CoGroup*, and *Cross*. DDP patterns enable programs to execute in parallel by splitting data in distributed computing environments. Originating from higher-order functional programming [7], each DDP pattern executes user-defined functions (UF) in parallel over input datasets. Since DDP execution engines often provide many features for execution, including parallelization, communication, and fault tolerance, application developers only need to select the appropriate DDP pattern for their specific data processing tasks, and implement the corresponding UFs.

The definitions and key properties of the DDP patterns are listed below and are illustrated in Fig. 1. Map and Reduce process a single input, while CoGroup, Match, and Cross process two inputs. For all patterns, each input data is a list of key-value pairs. These pairs are partitioned and shuffled based on the pattern definition and are sent to UF to process. One UF is instantiated for each independent input data and these UF instances can run in parallel. The formal definitions of these patterns can be found in [8].

- **Map:** Independently processes each key-value pair from the input. Consequently, the UF instances are called independently for the key-value pairs of the input list.
- **Reduce:** Partitions the key-value pairs by their keys. All pairs with the same key are grouped and handled together in one UF instance.
- **CoGroup:** Partitions the key-value pairs of the two input lists according to their keys and groups values for each key. For each input, all pairs with the same key form one list. So

[2] Cascading Project: http://www.cascading.org/, 2014

[3] Hadoop Project: http://hadoop.apache.org, 2014

[4] Stratosphere Project: http://www.stratosphere.eu, 2014

Digital Object Indentifier 10.1109/MCSE.2014.50 1070-9924/$26.00 2014 IEEE

each UF instance gets inputs as a key and two value lists for the key. If a key is only at one input list, the value list for this key from the other input will be empty.

- **Match:** Partitions the key-value pairs of the two input lists according to their keys without value grouping. A Cartesian product of the two value lists for the same key is generated, and each value pair from the product and the key will be processed by a separate UF instance. If a key is only in one input list, the UF will not be executed for this key.
- **Cross**: Processes all elements from the Cartesian product of the key-value pairs of the two input lists. Each UF instance processes one element from the Cartesian product and its inputs are two key-value pairs from the two input lists respectively.

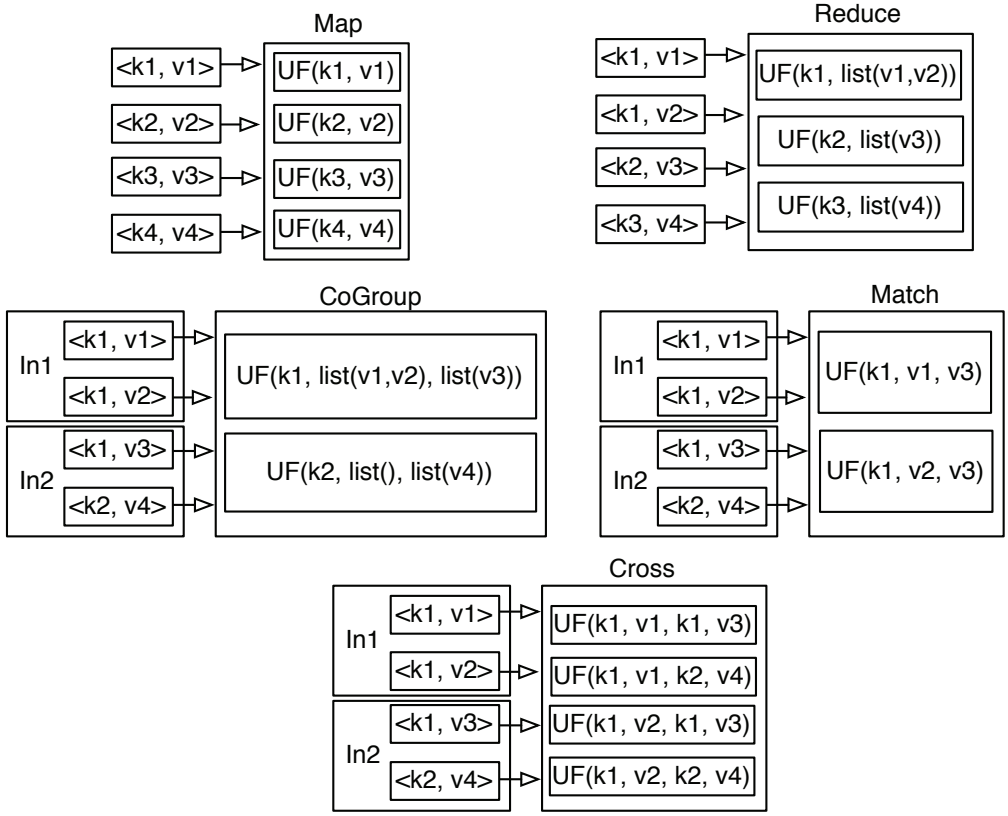


Fig. 1. Distributed Data Parallel Patterns to execute user-defined functions in parallel over input datasets.

Due to the increasing popularity and adoption of these DDP patterns, a number of execution engines have been implemented to support one or more of them. These DDP execution engines manage underlying communications and data transfers, and execute UF instances in parallel. When running on distributed resources, DDP engines can achieve good scalability and performance acceleration. Hadoop is the most popular MapReduce execution engine. Additionally, Cloud MapReduce [5] and MapReduce-MPI [6] can also run applications built by MapReduce patterns. The Stratosphere system [8] supports all of the DDP patterns illustrated in Fig. 1. The CoGroup, Match, and Cross may be transformed into Map and Reduce to run on Hadoop as described in Section 4.2. Since each DDP execution engine defines its own API for how UFs should be implemented, an application implemented for one engine may be difficult to

[5] Cloud MapReduce: http://code.google.com/p/cloudmapreduce/, 2014

[6] MapReduceMPI: http://mapreduce.sandia.gov/, 2014

Digital Object Indentifier 10.1109/MCSE.2014.50 1070-9924/$26.00 2014 IEEE

run on other engines. The visual programming approach employed by workflow systems can help to overcome this difficulty using the approach described in this article.

3. Actor-Oriented Programming and Kepler Scientific Workflow System

Unlike object-oriented programming where the basic elements are objects, actor-oriented programming is built on top of actors and inherits parallelization among actors. An actor provides an independent function, such as job submission or web service invocation. Actors can be connected via links that determine how data flows from one actor to another. Actors are categorized into *atomic* actors and *composite* actors, where composite actors, also called *sub-workflows,* are composed of atomic and/or other composite actors. In this way, we can achieve hierarchical modeling. Actor execution is data-driven: at runtime, each actor execution consumes a set of input data and generates a set of output data, and an actor repeatedly executes as long as it keeps receiving input data.

The Kepler open-source scientific workflow system is a cross-project collaboration to serve scientists from different disciplines. Kepler inherits from and extends Ptolemy II[7] to follow the actor-oriented paradigm to manage, process, and analyze scientific data. Kepler provides a graphical user interface (GUI) for designing, managing and executing scientific workflows, where each step is an actor. Actors are linked together to implement a computational solution to a scientific problem.

Actor-oriented programing decouples components from execution orchestration. Kepler and Ptolemy provide a special type of entity, called *Director*, to manage the execution of the linked actors. Each director specifies a model of computation that governs the interaction between actors. At runtime, the director determines when and how to send data to each actor and trigger its execution. Different directors have been defined in Kepler and Ptolemy to support different models of computation [9]. For instance, the SDF (Synchronous Data-Flow) director has fixed data production and consumption rates for each actor execution and the workflow execution is done in one thread. The PN (Process Network) director supports dynamic data production and consumption rates and each actor executes in its own thread.

The decoupling of actors and directors greatly enhances actor reuse with different directors. The same set of actors may be executed by different directors to have different behaviors. Further, the hierarchical structure of a workflow enables coexistence of multiple directors within one workflow. It makes the actor-oriented approach an ideal choice for heterogeneous Big Data applications where different steps have different scalability and execution needs. For instance, a sub-workflow that executes MPI jobs can be connected to a sub-workflow that executes MapReduce jobs within one workflow if proper directors are employed for the sub-workflows.

4. Actor-Oriented DDP Workflows in Kepler

4.1 Actors for DDP Patterns

The DDP framework fits actor-oriented programming very well. Since each DDP pattern expresses an independent higher-order function, we can define a separate DDP actor for each pattern. Unlike other actors, these higher-order DDP actors do not process the input data sent to them as a whole. Instead, they first partition the input data and then process each partition separately.

The UF for the DDP patterns is an independent component and can naturally be encapsulated within a DDP actor. The logic of the UF can either be expressed as a sub-workflow or compiled code. In the latter case, if users already implemented their UFs using the API for a DDP engine, they just need to configure this information in the DDP actor. Otherwise, users can compose a

[7] Ptolemy II website: http://ptolemy.berkeley.edu/ptolemyII/, 2014.

Digital Object Indentifier 10.1109/MCSE.2014.50 1070-9924/$26.00 2014 IEEE

sub-workflow for their UF in the Kepler GUI using specific auxiliary actors for the DDP pattern and any other general actors. Since the sub-workflow is not specific to any engine API, different DDP engines in Kepler can execute it. As with other actors, multiple DDP actors can be linked to construct bigger applications.

4.2 Director for DDP Patterns

Each DDP pattern defines its execution semantics, i.e., how data is partitioned and processed by the pattern. This clear definition enables decoupling between a DDP pattern and its execution engine. To execute DDP workflows on different DDP execution engines, we have implemented a DDP director in Kepler. Currently, this director can execute DDP workflows with either Hadoop or Stratosphere. At runtime, the director will detect the availability of Hadoop or Stratosphere execution engine and transform workflows into their corresponding jobs. The adaptability of the director makes it user-friendly since it hides the underlying execution engines from users.

All the DDP patterns described in Section 2 are supported by Stratosphere, and there is a one-to-one mapping between DDP workflows in Kepler and Stratosphere jobs. The DDP director can directly convert DDP workflows to jobs executable by Stratosphere. If a UF implementation code, and not a sub-workflow, is specified in DDP actors, the Stratosphere execution engine can invoke it directly. For DDP actors containing a sub-workflow, the Kepler engine is called with the partitioned data to process sub-workflows in the Stratosphere jobs. The DDP director also handles the necessary data type conversion between Kepler and Stratosphere during their interactions.

Table 1. Algorithm to transform a DDP workflow into Hadoop jobs

```
1.  List transformToHadoopJobs(workflow) {
2.    List sinkActorsList = getSinkActors(); //get the list of actors that have no downstream actors
3.    List hadoopJobList = new List();
4.    for (each actor in sinkActorsList) {
5.      hadoopJobList.add(getHJobs(actor, emptyList));
6.    }
7.  }

8.  List getHJobs(actor, tmpHJList) {
9.    if (actor is null)
10.     return tmpHJList; //return the current job list if there is no upstream actor.
11.   else if (actor is Map) {
12.     if (tmpHJList is empty) {
13.       Job job = new Job();
14.       job.setMapper(actor); //create a job whose Mapper runs the sub-workflow of the actor
15.     } else
16.       tmpHJList.getFirst().mergeMap(actor); // merge the Map into the current job.
17.     return getHJobs(actor.getUpstreamActor(), tmpHJList);
18.   } else { //actor could be Reduce, Match, CoGroup or Cross
19.     Job job = new Job();
20.     if (actor is Reduce) {
21.       job.setReducer(actor);
22.       tmpHJList.addToFront(job); //add job in front of the current jobs.
23.       return getHJobs(actor.getUpstreamActor(), tmpHJList);
24.     } else { //actor could be Match, CoGroup or Cross. All have two upstream actors.
25.       if (actor is Match) {
26.         job.setMapper(TaggingMapper); // TaggingMapper tags different values to the two
          inputs.
27.         job.setReducer(MatchReducer); // MatchReducer differentiates two inputs based tags
          and calls Match sub-workflow.
```

Digital Object Indentifier 10.1109/MCSE.2014.50 1070-9924/$26.00 2014 IEEE

```
28.         } else if (actor is CoGroup) {
29.             job.setMapper(TaggingMapper);
30.             job.setReducer(CoGroupReducer); // CoGroupReducer differentiates two inputs
          based tags and calls CoGroup sub-workflow.
31.         } else if (actor is Cross) {
32.             Job job2 = new Job();
33.             job2.setMapper(CacheWritterMapper); // CacheWritterMapper writes the first input
          data into Hadoop distributed cache.
34.             job.setMapper(CrossMapper); // CrossMapper gets the second input data from
          interface and the first input data from distributed cache, and calls Cross sub-workflow.
35.             tmpHJList.addToFront(job2);
36.         }
37.         tmpHJList.addToFront(job);
38.         tmpHJList.addToFront(getHJobs(actor.getFirstUpstreamActor(), tmpHJList)); //process
          the first upstream actor
39.         return getHJobs(actor.getSecondUpstreamActor(), tmpHJList); //process the second
          upstream actor and return
40.       }
41.     }
42. }
```

The DDP director can also execute DDP workflows on Hadoop. Since the CoGroup and Cross patterns are not supported by Hadoop, the director must convert these into Map and Reduce patterns. Unlike Map and Reduce patterns that process all input datasets in the same way, Match, CoGroup, and Cross differentiate inputs into two sets. Table 1 shows how to transform a DDP workflow into Hadoop jobs. It first finds all actors without any downstream actors (Line 2) and then traverses the workflow to add Hadoop jobs (Line 5). The function *getHJobs()* is called recursively until no upstream actors can be found. For Map actors, a new Hadoop job is generated with the Map actor if there are no Hadoop jobs yet (Line 14). Otherwise, the Map actor is merged into the current first Hadoop job (Line 16). By merging consecutive Map actors into one Hadoop job, the algorithm minimizes data staging on the Hadoop Distributed File System among the Map actors. For other DDP actors, a new Hadoop job has to be generated since output data needs cross-node shuffling before the next job. For Match and CoGroup, a special Map task is used to tag input values to indicate that they are from different sources (Lines 26 and 29). Next, a special Reduce task is used to split its input data into two lists based on their value tags, and calls the Kepler engine to execute the sub-workflow defined in the Match or CoGroup actor based on the semantics of the corresponding pattern (Lines 27 and 30). However, this approach does not work for Cross since the inputs of each Cross UF instance may have different keys. Two Hadoop jobs are generated for each Cross actor. The first job writes the first input dataset into Hadoop's Distributed Cache using a Map (Line 33). Next, the second job reads the second input dataset, and the first from the Distributed Cache, and executes the Cross sub-workflow with the combined key-value pairs.

4.3 Support Multiple Alternative DDP Choices

While each DDP pattern focuses on a type of data parallel execution, there might be multiple or combinations of DDP patterns suitable to accelerate execution of one specific tool [10]. It brings a challenge on how to enable end users to easily select and switch DDP patterns. In our ongoing bioKepler project [8] [11], we address this challenge by implementing a special actor in Kepler, called *Execution Choice,* to include all possible DDP patterns as sub-workflows and

[8] bioKepler website: http://www.biokepler.org/ 2014.

Digital Object Indentifier 10.1109/MCSE.2014.50 1070-9924/$26.00 2014 IEEE

provide an easy way to switch the patterns. The Execution Choice actor also facilitates connecting with other components to build bigger applications. Fig. 2 shows the configuration dialogue for the actor where the *Choice* parameter lists three available DDP pattern combinations for a task. The sub-workflows for this actor are in different tabs of the actor and their details will be discussed in Section 6.2. End users only need to make the proper DDP choices for their executions by setting the Choice parameter.

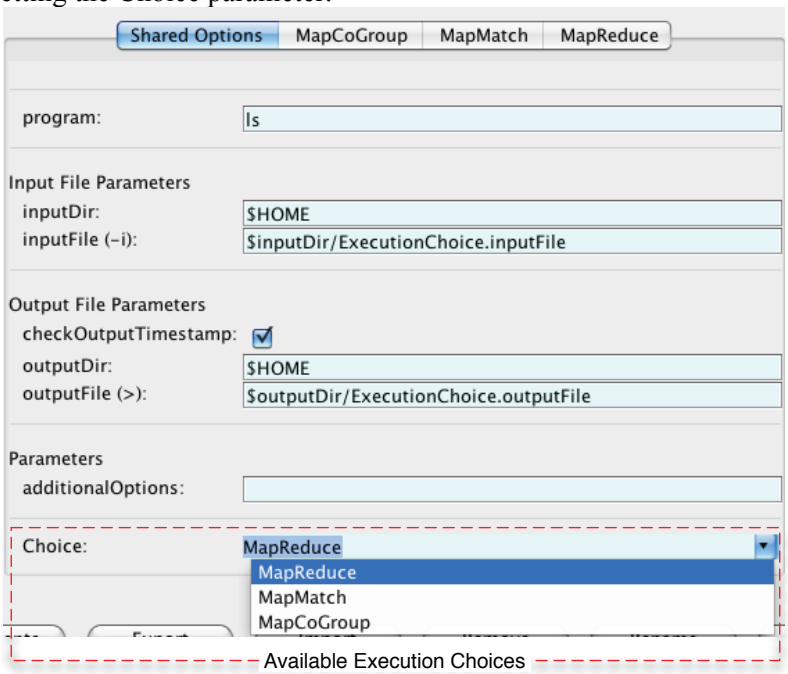


Fig. 2. Multiple alternative DDP choices in Kepler via Execution Choice actor.

5. Performance Analysis for Legacy Tool Parallelization in DDP

The DDP patterns provide a simple and efficient way to parallelize standalone legacy tools if their executions fit the patterns. One DDP job can spawn many worker tasks that run in parallel on distributed nodes. Yet there are many configuration factors affecting execution performance. We identify three factors and analyze their relationships in order to know how to get the best performance based on the information of legacy tools, input data, and available computation resources.

The first configuration factor is how to partition the data based on available computation resource information. The default data partition policy in Hadoop splits data into 64MB blocks and tries to do MapReduce processing in parallel for the blocks. Our experiences with many legacy tools, mostly from the bioinformatics domain, show load balancing is often more important when parallelizing legacy tools since each execution of the legacy tool may take a very long time even when the input data is small. So we define a parameter in our DDP actors, called *ParallelNum*, to tell how many partitions the input data should be split into. By this way, we can get even data partitions and balanced execution if the number is a multiple of the available worker node number.

The second factor is the data size to be processed by the legacy tool for each of its execution. The DDP implementations of Big Data analytics applications often try to process minimal input data, such as one line, for each execution of the DDP UF. We find this approach to be inefficient mainly due to the overhead of running legacy tools. Each legacy tool needs a separate process/thread to run. If each execution only processes minimal input data, we will get maximal legacy tool execution times. Too many legacy tool execution times could cause a lot of overhead for loading the tool and input data. In our experience, the execution performance will be better

Digital Object Indentifier 10.1109/MCSE.2014.50 1070-9924/$26.00 2014 IEEE

when we send relatively large input for each tool execution. In our DDP framework, we define specific parameters for each data format, which tells how much data will be sent to each tool execution.

The third factor is resource allocation for worker tasks and legacy tool execution. Because the legacy tools execute in separate processes/threads, besides resource (CPU, memory, etc.) allocated for worker tasks for distributed job and data management, we also need to allocate resources for legacy tool executions. At each worker node, each worker task will keep invoking the legacy tool until all the data split assigned for the worker task is processed. But at any time, each worker task will only have one running process for the legacy tool. We will explore the proper relationship between available core number and worker task number in the experiment section.

The relationships between the above factors are as follows. The value of the ParallelNum parameter will be same with the split number of the input data and the number of worker tasks. Each worker task will process one data split whose size is the quotient of the total data size by ParallelNum. Inside of each worker task, the execution number of legacy tool is the ceiling of the quotient of data split size by data size per execution. For instance, if input data has forty lines and the ParallelNum value is four, there will be four parallel worker tasks and each job processes ten lines. Each parallel worker task calls the legacy tool to process its data. If we set the line number for each the legacy tool execution to be six, each parallel worker task will call the legacy tool twice: six line for the first execution and four lines for the second. During the execution, there will be four worker tasks and four external processes for legacy tool running in parallel.

6. Applications in Bioinformatics

We are applying the DDP approach in bioinformatics to parallelize existing community-built bioinformatics tools. We first identify execution bottlenecks of a bioinformatics tools for its scalability. If a tool can be parallelized via DDP patterns, we wrap the tool within suitable DDP actors based on the identified execution requirements. Among the 42 bioinformatics tools we investigated so far, 14 of them can be parallelized using one or more DDP patterns. We will explain two use cases as examples of this study. The first one re-implements the RAMMCAP workflow [12] in Kepler to demonstrate how to build large-scale DDP applications in Kepler. The second one is built on top of CloudBurst [13] to show how multiple DDP patterns can work to parallelize the execution of the same tool.

6.1 RAMMCAP

The RAMMCAP (Rapid Analysis of Multiple Metagenomes with a Clustering and Annotation Pipeline) workflow addresses the computational challenges imposed by the huge size and large diversity of metagenomic data [12]. RAMMCAP includes many bioinformatics tools for different functions, including function annotation, clustering, and open reading frame prediction. Only some of the tools can be parallelized using DDP patterns, which makes it difficult to build such an application directly using Hadoop or Stratosphere. Since Kepler supports multiple models of computations within one workflow, it is easy to build and execute RAMMCAP in Kepler.

Fig. 3 shows a simplified RAMMCAP workflow in Kepler. The hierarchical structure shows different directors are employed at different layers to coordinate actor execution accordingly. The workflow includes nine bioinformatics tools where three of them can be parallelized, namely, tRNAscan-SE, rpsblast_for_COG and rpsblast_for_KOG. The parallelization is done by first partitioning input data, then running the tool in parallel with partitioned inputs, and merging results in the end. It fits the semantics of DDP Map pattern, so we employ the Map actor. Data partitioning and merging is configured in DDP DataSource and DataSink actors. The sub-workflow inside of the Map actor first reads a fraction of input data sent by execution engine, and

Digital Object Indentifier 10.1109/MCSE.2014.50 1070-9924/$26.00 2014 IEEE

then calls the bioinformatics tool with the data fraction. At runtime, the DDP director coordinates with the available DDP execution engine to enable data partitioning/merging and parallel processing of the sub-workflow inside of the Map actors.

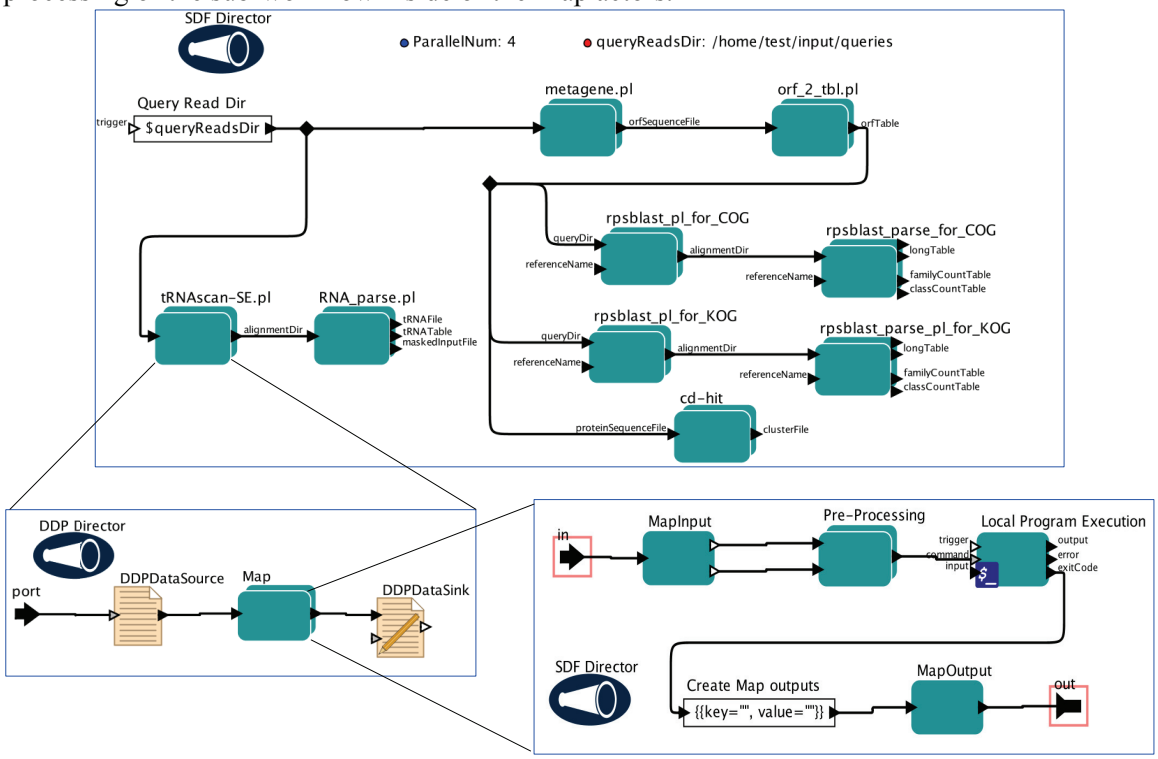
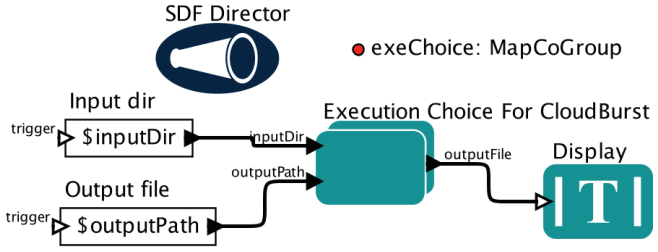


Fig. 3. RAMMCAP workflow in Kepler.

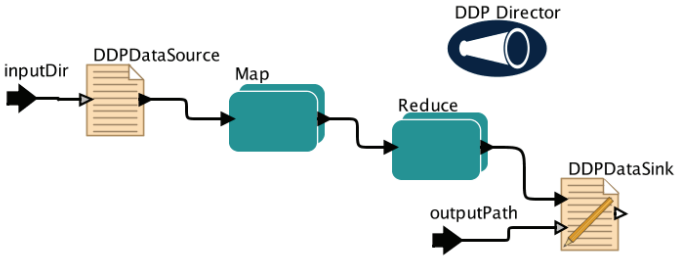
6.2 CloudBurst

CloudBurst is a parallel sequence-mapping tool, which maps query sequences to a reference sequence dataset to find and locate similar fragments in reference data for each query sequence. This tool is originally implemented using MapReduce. It has two input datasets and processes them differently. So the two datasets have to be distinguished internally throughout the application. We find it is more straightforward and simpler to separate the two datasets at DDP pattern level. By using Match or CoGroup, we can not only have separate inputs inside of Match/CoGroup UF, but also use two Maps before Match/CoGroup to process the two data sets separately. So CloudBurst could be executed using MapReduce, MapMatch, or MapCoGroup pattern combinations [10]. Fig. 4 shows the top-level workflow and the sub-workflow for each combination. All these are sub-workflows of the Execution Choice actor for CloudBurst shown in Fig. 2. Since our re-implementation of CloudBurst already has the Java classes for each pattern, we just need to configure the DDP actors to use these classes. The GUI of each sub-workflow also clearly depicts the logic of each DDP pattern combination. At runtime, the DDP Director will get the sub-workflow selected by users at the Execution Choice actor, transform it, and execute it via a DDP execution engine.

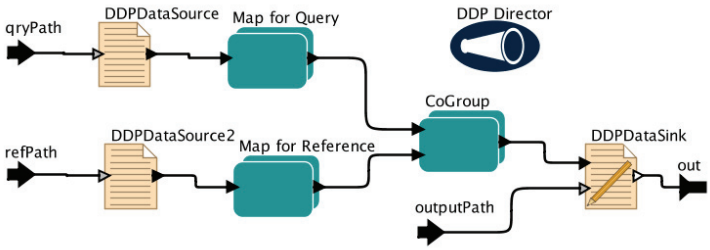


Digital Object Indentifier 10.1109/MCSE.2014.50 1070-9924/$26.00 2014 IEEE

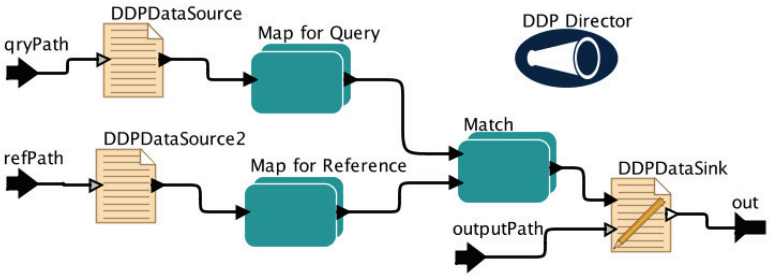
(a). Top-level workflow of CloudBurst.



(b) MapReduce Sub-workflow of CloudBurst.



(c). MapCoGroup Sub-workflow of CloudBurst.



(d). MapMatch Sub-workflow of CloudBurst.

Fig. 4. CloudBurst workflow in Kepler with Multiple DDP Implementations.

7. Execution Experiments and Analysis

7.1 Scalability

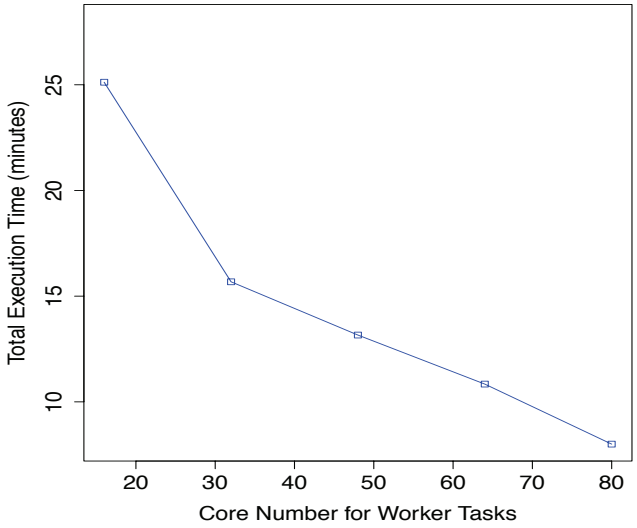
We performed several experiments to measure the scalability of the RAMMCAP workflow and analyze its performances. The input dataset has nine million sequences. The experiments are done using six compute nodes in a cluster environment, where each node has two eight-core 2.6GHz CPUs, and 64GB of memory. Each node could access the sequence data, and the bioinformatics tools via a shared file system. The tests were done with Hadoop version 0.22. In the tests, one node is assigned for task coordination and others for worker tasks.

We first tested the scalability of the workflow with the ParallelNum value always half of the available worker core numbers. The experiment results, shown in Fig. 5 (a), tell that the execution has good scalability when we increase the available cores. The reason that the speedup ratio is less when running on more cores is twofold. First, the communications between the cores will be more complex in a larger environment and cause more overhead. Second, the workflow has several tools that cannot run in parallel, these sequential steps have a bigger impact on speedup ratio in a larger environment.

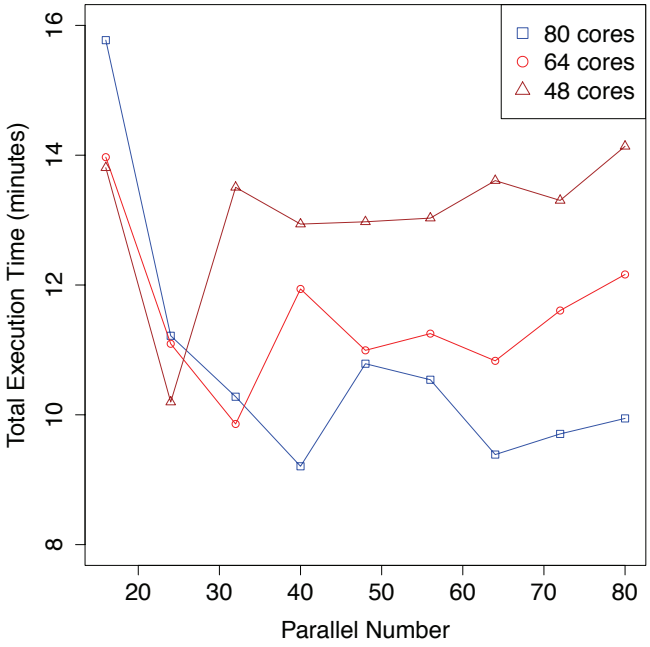
To understand how the factors described in Section 5 affect workflow performance, we ran the workflow with different ParallelNum and data sizes per execution (called *SeqNumPerExe*),

Digital Object Indentifier 10.1109/MCSE.2014.50 1070-9924/$26.00 2014 IEEE

values on different worker cores. The experiments results in Fig. 5 (b)-(c) show that: 1) Both ParallelNum and SeqNumPerExe are important factors in terms of performance; 2) Execution time first decreases when ParallelNum value increases, and then fluctuates; the best ParallelNum value is always half available core number; 3) Execution time decreases when the SeqNumPerExe value increases until some point. After that, increasing data size per execution does not affect the execution time much.

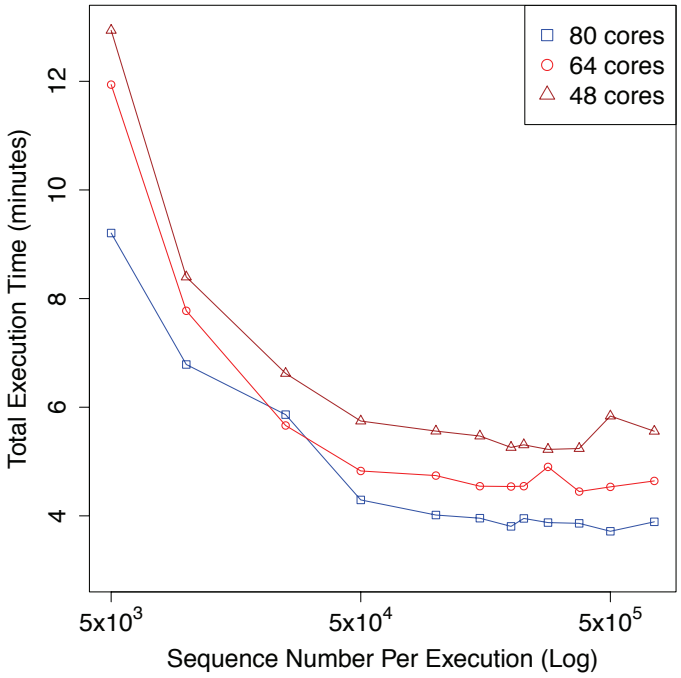


(a) Performance with different available CPU cores.



(b) Performance changes with different parallel number.

Digital Object Indentifier 10.1109/MCSE.2014.50 1070-9924/$26.00 2014 IEEE



(c) Performance with different sequence number per execution (Log scale).

Fig. 5. RAMMCAP workflow scalability experiment and performance analysis.

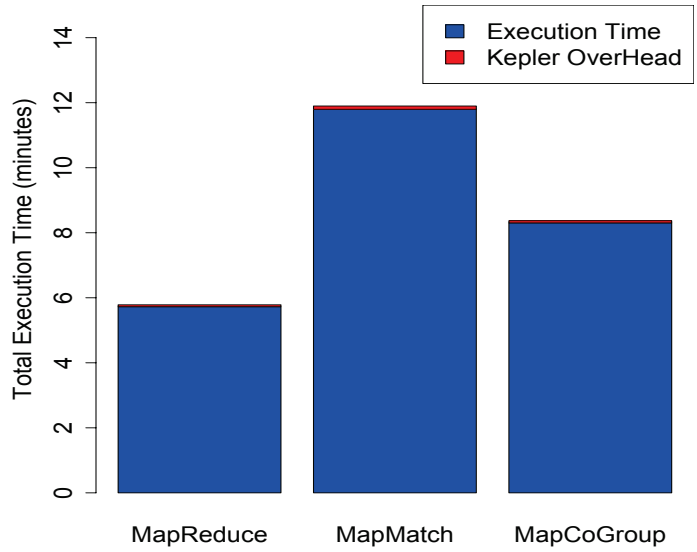
7.2 Overhead with Kepler Workflow System

We also measured the overhead of executing DDP applications as Kepler workflows with the same query dataset and execution environment. We compared the performance of the CloudBurst workflow implemented in Kepler (shown in Fig. 2 and 5) with its native Java implementation. The reference dataset has over 1.2 million sequences. To check the overhead relation with data size, we also test with the same reference dataset and only first 5000 sequences for query dataset. The tests were done with Stratosphere version 0.2. The UFs for Reduce, CoGroup, and Match were executed from three thousand to eleven billion times.

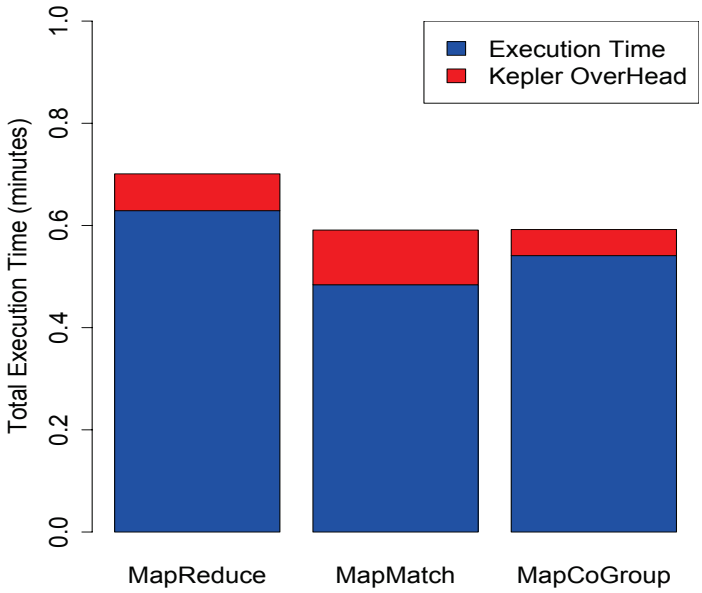
All of the experiments below were performed with ParallelNum value as 40. Since all of the available DDP pattern options are in one workflow, we only need to run the same DDP workflow. The choice parameter value is different for each execution in order to specify which DDP sub-workflow will be executed.

The experimental results are shown in Fig. 6. From the figure, we can see the overhead is relatively constant, ranging from 0.05 to 0.1 minutes for both small and large query data. Most of this overhead lies in Kepler instantiation and workflow parsing, which is independent from the input dataset sizes. It verifies that Kepler facilitates DDP application development with small execution overhead. The reason of execution time differences for the three DDP combinations is analyzed in [10].

Digital Object Indentifier 10.1109/MCSE.2014.50 1070-9924/$26.00 2014 IEEE



(a) Kepler overhead for large input data.



(b) Kepler overhead for small input data.

Fig. 6. Kepler Overhead for CloudBurst application.

8. Conclusions and Future Work

We presented a new and innovative visual programming approach to scale Big Data applications on data parallel execution engines, e.g., Hadoop and Stratosphere, by implementing distributed data parallel execution patterns as actors. We showed application of the presented approach to next-generation sequence analysis in bioinformatics. The application demonstrated how the developed programming techniques could be used to achieve data parallel scalability for some sequence analysis tools while using them in combination with other sequential tools within workflows. This approach enables users of such scientific tools (and other user functions) to build and execute applications without the need for writing wrapping scripts using the APIs provided by different execution engines. Instead, our approach presents a more generalized way to build

Digital Object Indentifier 10.1109/MCSE.2014.50 1070-9924/$26.00 2014 IEEE

the data parallel UFs as sub-workflows in Kepler and execute them on different data parallel execution engines. Although the presented approach adds a minor overhead to the execution time compared to the scripted wrappers as concluded by the experiments in Section 7.2, its flexibility to switch between and mix-and-match multiple DDP patterns and engines has the potential to actually reduce the execution time when the correct patterns are selected. We also analyzed and verified three factors affecting executing legacy tools in our DDP framework. Additionally, the presented approach enables combining the DDP tasks with local tasks and tasks that require other forms of parallelism within one workflow. In summary, the contributions of this article result in improved programmability and scaling flexibility of Big Data applications while enabling applications that are built on the strengths of different parallelization methods without writing wrapping or bash scripts.

For future work, we will further study load balancing of DDP patterns and how performance changes based on data location and replication in HDFS and speculative task launching. We will also apply our approach in more applications and test them in larger environments. In addition, we plan to improve our approach by profiling user applications and measuring input data to enable automatic execution optimization.

9. Acknowledgments

The authors would like to thank the rest of Kepler and bioKepler teams for their collaboration. This work was supported by NSF ABI Award DBI-1062565 for bioKepler.

References

[1] L. Atzori, A. Iera, G. Morabito, “The Internet of Things: A survey”, Computer Networks, 54 (15), October 2010, pp. 2787-2805, ISSN 1389-1286, http://dx.doi.org/10.1016/j.comnet.2010.05.010.

[2] I. J. Taylor, E. Deelman, D. B. Gannon, M. Shields (Eds.), Workflows for e-Science: Scientific Workflows for Grids. Springer, December 2006.

[3] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” Communications of the ACM 51 (1) (2008), pp. 107-113.

[4] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur, “Oozie: towards a scalable workflow management system for Hadoop.” In Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies, p. 4. ACM, 2012.

[5] 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.

[6] 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), pp. 1039-1065.

[7] B. J. MacLennan, M. Mayer, R. Redhammer. Functional programming: practice and theory. Addison-Wesley, 1990.

[8] 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.

[9] A. Goderis, C. Brooks, I. Altintas, E. A. Lee, C. Goble, “Heterogeneous composition of models of computation, Future Generation Computer Systems”, 25(5), May 2009, pp. 552-560, http://dx.doi.org/10.1016/j.future.2008.06.014.

[10] J. Wang, D. Crawl, I. Altintas, K. Tzoumas, V. Markl, “Comparison of Distributed Data-Parallelization Patterns for Big Data Analysis: A Bioinformatics Case Study”, In Proceedings of the Fourth International Workshop on Data Intensive Computing in the Clouds (DataCloud), 2013.

[11] 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.

Digital Object Indentifier 10.1109/MCSE.2014.50 1070-9924/$26.00 2014 IEEE

[12] W. Li. "Analysis and comparison of very large metagenomes with fast clustering and functional annotation", BMC Bioinformatics, 10:359, 2009.
[13] M. Schatz, "CloudBurst: Highly sensitive read mapping with MapReduce", Bioinformatics 25 (11) (2009), pp. 1363-1369.

Jianwu Wang, Ph.D., is the Assistant Director for Research at the Workflows for Data Science (WorDS) Center of Excellence at San Diego Supercomputer Center (SDSC), University of California, San Diego (UCSD), and an Assistant Project Scientist at SDSC, UCSD. He is also an Adjunct Professor at North China University of Technology, China. He is the main researcher for distributed architectures of the open-source Kepler Scientific Workflow System. His research interests include Big Data, Scientific Workflow, Service-Oriented Computing, End-User Programming and Distributed Computing. Wang received his Ph.D. in Computer Science from the Chinese Academy of Sciences, China. Contact him at jianwu@sdsc.edu.

Daniel Crawl, Ph.D., is the Assistant Director for Development at the Workflows for Data Science Center of Excellence at SDSC, UCSD. He is the lead architect for the overall integration of distributed data parallel execution patterns and the Kepler Scientific Workflow System. He conducts research and development of execution patterns, bioActors, and distributed directors. Crawl received his Ph.D. in Computer Science from the University of Colorado at Boulder, U.S. Contact him at crawl@sdsc.edu.

Ilkay Altintas, Ph.D., is the Director for the Workflows for Data Science Center of Excellence at SDSC, UCSD. She is a co-initiator of and an active contributor to the Kepler Scientific Workflow System, and the co-author of publications related to eScience at the intersection of scientific workflows, provenance, distributed computing, bioinformatics, observatory systems, conceptual data querying, and software modeling. Altintas received her Ph.D. in Computer Science from the University of Amsterdam, the Netherlands. Contact her at altintas@sdsc.edu.

Weizhong Li, Ph.D., is an Associate Research Scientist at Center for Research in Biological Systems, UCSD. He has a background in computational biology and bioinformatics. His research focuses on developing computational methods for sequence, genomic and metagenomic data analysis. Li received his Ph.D. in Computational Chemistry, Nankai University, China. Contact him at liwz@sdsc.edu.

Complete contact information
mailing address [for shipping your complimentary copy of the publication], phone/fax, email of each author

Jianwu Wang
San Diego Supercomputer Center, University of California, San Diego
9500 Gilman Drive, MC 0505, La Jolla, CA 92093-0505, U.S.A.
Phone : 858-534-5110
Email : jianwu@sdsc.edu

Dan Crawl
San Diego Supercomputer Center, University of California, San Diego
9500 Gilman Drive, MC 0505, La Jolla, CA 92093-0505, U.S.A.

Digital Object Indentifier 10.1109/MCSE.2014.50 1070-9924/$26.00 2014 IEEE

Phone : 858-822-3694
Email : crawl@sdsc.edu

Ilkay Altintas

San Diego Supercomputer Center, University of California, San Diego
9500 Gilman Drive, MC 0505
La Jolla, CA 92093-0505, U.S.A.

Phone : 858-822-5453
Email : altintas@sdsc.edu

Weizhong Li

Center for Research in Biological Systems, University of California San Diego
9500 Gilman Drive MC 0446, Atkinson Hall, Room 3113
La Jolla CA 92093-0446, U.S.A.

Phone: (858)-534-4143
Email : liwz@sdsc.edu

Digital Object Indentifier 10.1109/MCSE.2014.50 1070-9924/$26.00 2014 IEEE