Big Data Applications Using Workflows for Data Parallel Computing

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

In the Big Data era, workflow systems must embrace data parallel computing techniques for efficient data analysis and analytics. Here, an easy-to-use, scalable approach is presented to build and execute Big Data applications using actor-oriented modeling in data parallel computing. Two bioinformatics use cases for next-generation sequencing data analysis demonstrate the approach's feasibility.

As the Internet of Things and other data acquisition and generation technologies advance, the amount of data being generated is growing at an exponential rate at all scales in many online and scientific platforms. This mostly unstructured and variable data 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 the natural sciences to marketing research. To analyze and digest such heterogeneous data, we need 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 the applications being developed. These applications typically involve data ingestion, preparation (for example, extract, transform, and load), integration, analysis, visualization, and dissemination—referred to as data science workflows. Developing a data science workflow involves combining data and processes into a configurable, structured set of steps that implement automated computational solutions of an application with such capabilities as provenance and execution management, reporting tools, and integration of distributed computation and data management technologies. 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 with and facilitate integration of other tools. New programming techniques are needed for building effective and scalable solutions spanning data science workflows. The flexibility of workflow systems to combine tools and data make them an ideal choice for developing data science applications involving common Big Data programming patterns.

Big Data workflows have been an active research area since the introduction of scientific workflows. After the development and general adoption of MapReduce as a Big Data programming pattern, several workflow systems were built or extended to enable programmability of MapReduce applications. These systems include Oozie, Nova, Azkaban (azkaban.github.io), and Cascading (www.cascading.org). Here, we present programming extensions of MapReduce and other Big Data programming patterns to the Kepler Workflow Environment (kepler-project.org) and the engines built on top of the Hadoop (hadoop.apache.org) and Stratosphere (www.stratosphere.eu) systems that allow them to execute workflows using these patterns.

The tools presented in this article are applicable to Big Data workflows in all domains. By leveraging the workflow composition and management capabilities of Kepler, and the execution characteristics of distributed data parallel (DDP) patterns such as MapReduce, we provide a general 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.
The approach scales to structured, nonstructured, and semistructured Big Data, and is applicable on a range of computing resources including Hadoop clusters, XSEDE, and Amazon’s Elastic Compute Cloud (EC2).

Data Parallel Computing in Distributed Environments

Several design structures are commonly used in data parallel analysis and analytics applications. To generalize and reuse these design structures in more applications, DDP patterns have been identified to easily build efficient data parallel applications. These DDP patterns include Map, Reduce, CoGroup, and Cross. DDP patterns allow 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 (UFs) in parallel over input datasets. Because DDP execution engines often provide many features for execution, including parallelization, communication, and fault tolerance, application developers need only select the appropriate DDP pattern for their specific data processing tasks and implement the corresponding UFs.

Figure 1 illustrates the DDP patterns, which are defined as follows:

- **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, each UF instance gets inputs as a key and two value lists for the key. If a key is only at one input

---

**Figure 1.** Distributed data parallel (DDP) patterns to execute user-defined functions in parallel over input datasets.
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 won’t 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.

Map and Reduce process a single input, whereas 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 instances to process. One UF is instantiated for each independent input data and these UF instances can run in parallel. Formal definitions of these patterns are available elsewhere.

Because of the increasing popularity and adoption of these DDP patterns, several 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. Cloud MapReduce (code.google.com/p/cloudmapreduce) and MapReduce-MPI (mapreduce.sandia.gov) can also run applications built using MapReduce patterns. The Stratosphere system supports all of the DDP patterns illustrated in Figure 1. The CoGroup, Match, and Cross patterns can be transformed into Map and Reduce to run on Hadoop. Because each DDP execution engine defines its own API for implementing UF instances, it might be difficult to run an application implemented for one engine on other engines. As we show, the visual programming approach employed by workflow systems can help overcome this difficulty.

**Actor-Oriented Programming and the 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 a 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 as atomic or composite, 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 executes repeatedly 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 (ptolemy.berkeley.edu/ptolemyII) 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 programming decouples components from execution orchestration. Kepler and Ptolemy provide a special type of entity, called a **director**, to manage the execution of 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. A director is a model of computation that governs the interaction between actors. Typical directors include a 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 as atomic or composite, 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 executes repeatedly 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 (ptolemy.berkeley.edu/ptolemyII) 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 programming decouples components from execution orchestration. Kepler and Ptolemy provide a special type of entity, called a **director**, to manage the execution of 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 computation models. For instance, the synchronous dataflow director has fixed data production and consumption rates for each actor execution, and the workflow execution is done in one thread. The 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 can be executed by different directors to have different behaviors. Further, a workflow’s hierarchical structure enables the coexistence of multiple directors within one workflow. It makes the actor-oriented approach an ideal choice for heterogeneous Big Data applications in which different steps have different scalability and execution needs. For instance, a sub-workflow that executes message passing interface (MPI) jobs can be connected to a sub-workflow that executes...
MapReduce jobs within one workflow if proper directors are used for sub-workflows.

**Actor-Oriented DDP Workflows in Kepler**

The DDP framework is well-suited to actor-oriented programming. Because 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 input data as a whole. Instead, they partition the input data and then process each partition separately.

The UF for DDP patterns is an independent component and can naturally be encapsulated within a DDP actor. The logic of the UF can be expressed as a sub-workflow or as 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 sub-workflow for their UF in the Kepler GUI using specific auxiliary actors for the DDP pattern and any other general actors. Because the sub-workflow isn’t 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.

Each DDP pattern defines its execution semantics—that is, how it partitions and processes data. This clear definition enables decoupling between a DDP pattern and its execution engine. To execute DDP workflows on different DDP execution engines, we 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 the Hadoop or Stratosphere execution engines and transform workflows into their corresponding jobs. The director’s adaptability makes it user friendly because it hides the underlying execution engines from users.

Stratosphere supports all of the DDP patterns described in this article, and there’s 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.

The DDP director can also execute DDP workflows on Hadoop. Because the CoGroup and Cross patterns are not supported by Hadoop, the director must convert them into Map and Reduce patterns. Unlike Map and Reduce patterns, which process all input datasets in the same way, Match, CoGroup, and Cross split inputs into two sets.

Figure 2 shows how the director transforms a DDP workflow into Hadoop jobs. First, it finds all actors without any downstream actors (line 2) and then traverses the workflow to add Hadoop jobs (line 5). It calls the function `getHJobs()` 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 must be generated because 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’re 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 doesn’t work for Cross because the inputs of each Cross UF instance might have different keys. So, the director generates two Hadoop jobs 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.

Although each DDP pattern focuses on a type of data parallel execution, there might be multiple or combinations of DDP patterns suitable for accelerating execution of one specific tool. This brings a challenge on how to enable users to easily select and switch DDP patterns. To address this challenge, our ongoing bioKepler project (www.biopheleter.org) implements an execution choice actor in Kepler to include all possible DDP patterns as sub-workflows and provide an easy way to switch the patterns. The execution choice actor
Figure 2. Algorithm to transform a DDP workflow into Hadoop jobs.

```java
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) return tmpHJList; //return the current job list if there is no upstream actor.
10.  else if (actor is Map) {
11.     if (tmpHJList is empty) {
12.       Job job = new Job();
13.       job.setMapper(actor); //create a job whose Mapper runs the sub-workflow of the actor
14.     } else
15.     tmpHJList.getFirst().mergeMap(actor); // merge the Map into the current job.
16.     return getHJobs(actor.getUpstreamActor(), tmpHJList);
17.   } else if (actor could be Reduce, Match, CoGroup or Cross) {
18.     Job job = new Job();
19.     if (actor is Reduce) {
20.       job.setReducer(actor);
21.       tmpHJList.addToFront(job); //add job in front of the current jobs
22.     return getHJobs(actor.getUpstreamActor(), tmpHJList);
23.    } else if (actor is Match) {
24.       job.setReducer(MatchReducer); // MatchReducer differentiates two inputs based tags and calls Match sub-workflow.
25.     } else if (actor is CoGroup) {
26.       job.setReducer(CoGroupReducer); // CoGroupReducer differentiates two inputs based tags and calls CoGroup sub-workflow.
27.     } else if (actor is Cross) {
28.       Job job2 = new Job();
29.       job2.setReducer(CacheWarningMapper); // CacheWarningMapper writes the first input data into Hadoop distributed cache.
30.       job.setReducer(CrossMapper); // CrossMapper gets the second input data from interface and the first input data from distributed cache, and calls Cross sub-workflow.
31.     tmpHJList.addToFront(job2);
32. } else if (actor is Cross) {
33.     job.setReducer(CrossMapper); // CrossMapper gets the second input data from interface and the first input data from distributed cache, and calls Cross sub-workflow.
34.     tmplList.addToFront(job2);
35.   }
36.   return getHJobs(actor.getFirstUpstreamActor(), tmpHJList); //process the first upstream actor
37. }
38. }
```
also facilitates connecting with other components to build bigger applications. Figure 3 shows the configuration dialogue for the actor in which 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 (their details are discussed later). Users only need to make the proper DDP choices for their executions by setting the choice parameter.

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 many configuration factors affect execution performance. We identify three factors and analyze their relationships to determine how to get the best performance based on the information of legacy tools, input data, and available computation resources.

The first configuration factor is partitioning the data based on available computation resource information. The default data partition policy in Hadoop splits data into 64-Mbyte blocks and tries to perform MapReduce processing in parallel for the blocks. Our experiences with many legacy tools, mostly from the bioinformatics domain, show that load balancing is often more important when parallelizing legacy tools because each execution of the legacy tool can take a long time even when the input data is small. We therefore define a ParallelNum parameter in our DDP actors to indicate the number of partitions the input data should be split into. In this way, we get even data partitions and balanced execution if the number is a multiple of the number of available workers.

The second factor is the data size to be processed by the legacy tool for each 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 because of 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’ll 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, execution performance is better when we send relatively large input for each tool execution. In our DDP framework, we define specific parameters for each data format that tell 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, in addition to allocating resources (CPU, memory, and so on) to worker tasks for distributed job and data management, we also need to allocate resources for legacy tool execution. At each worker node, each worker task will continue invoking the legacy tool until all the data split assigned to the worker task is processed. But at any time, each worker task will only have one running process for the legacy tool. We explore the proper relationship between available core number and worker task number later.

The relationships between these factors are as follows. The value of the ParallelNum parameter will be the same as 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 each worker task, the execution number of the legacy tool is the ceiling of the quotient of data split size by data size per execution. For instance, if the input data has 40 lines and the ParallelNum value is 4, there will be four parallel worker tasks and each job will process 10 lines. Each parallel worker task calls the legacy tool to process its data. If we
set the line number for each legacy tool execution to be 6, each parallel worker task will call the legacy tool twice: six lines 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 tools running in parallel.

Applications in Bioinformatics
We’re applying the DDP approach in bioinformatics to parallelize existing community-built bioinformatics tools. We first identify the execution bottlenecks of a bioinformatics tool 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’ve investigated so far, 14 can be parallelized using one or more DDP patterns, which makes it difficult to build such an application directly using Hadoop or Stratosphere. Because Kepler supports multiple models of computations within one workflow, it’s easy to build and execute RAMMCAP in Kepler.

Figure 4 shows a simplified RAMMCAP workflow in Kepler. The hierarchical structure shows the different directors employed at different layers to coordinate actor execution. The workflow includes nine bioinformatics tools. Three of them—tRNAscan-SE, rpsblast_for_COG, and

Figure 4. Rapid Analysis of Multiple Metagenomes with Clustering and Annotation Pipeline (RAMMCAP) workflow in Kepler.
rpsblast_for_KOG—can be parallelized. 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 the DDP Map pattern, so we use the Map actor. Data partitioning and merging is configured in DDP DataSource and DataSink actors. The sub-workflow inside the Map actor first reads a fraction of input data sent by the execution engine, and 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 the Map actors.

CloudBurst
CloudBurst is a parallel sequence-mapping tool that maps query sequences to a reference sequence dataset to find and locate similar fragments in reference data for each query sequence. This tool was originally implemented using MapReduce. It processes its two input datasets differently, so the two datasets must be distinguished internally throughout the application. We find that it's more straightforward to separate the two datasets at the DDP pattern level. By using Match or CoGroup, we not only can have separate inputs inside the Match/CoGroup UFs, but can also use two Maps before Match/CoGroup to process the two datasets separately. So, CloudBurst could be executed using MapReduce, MapMatch, or MapCoGroup pattern combinations.

Figure 5 shows the top-level workflow and the sub-workflow for each combination. All of these are sub-workflows of the execution choice actor for CloudBurst shown in Figure 3. Because our re-implementation of CloudBurst already has the Java classes for each pattern, we only 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 receives the sub-workflow selected by users at the execution choice actor, transforms it, and executes it via a DDP execution engine.

Execution Experiments and Analysis
We performed several experiments to measure the scalability of the RAMMCAP workflow and analyze its performance. The input dataset has 9 million sequences. We conducted the experiments using six compute nodes in a cluster environment, where each node has two eight-core 2.6-GHz CPUs, and 64 Gbytes 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 to task coordination and others to worker tasks.

We first tested the workflow's scalability with the ParallelNum value always set to half of
available worker core numbers. The results, shown in Figure 6a, suggest 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 communication between cores will be more complex in a larger environment and cause more overhead. Second, the workflow has several tools that can’t run in parallel; these sequential steps have a bigger impact on speedup ratio in a larger environment.

To understand how the factors described previously affect workflow performance, we ran the workflow with different ParallelNum and data sizes per execution (called SeqNumPerExe), values on different worker cores. As Figures 6b and 6c show, both ParallelNum and SeqNumPerExe are important factors in terms of performance. Execution time first decreases when ParallelNum value increases, and then fluctuates; the best ParallelNum value is always half the available core number. In addition, execution time decreases when the SeqNumPerExe value increases until some point. After that point, increasing the data size per execution doesn’t affect the execution time much.

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 (Figures 3 and 6) with its native Java implementation. The reference dataset has more than 1.2 million sequences. To check the overhead relation with data size, we also tested with the same reference dataset and only the first 5,000 sequences for each query dataset. We conducted the tests using Stratosphere version 0.2. The UFs for Reduce, CoGroup, and Match were executed from 3,000 to 11 billion times.

All of the experiments were performed with the ParallelNum value set to 40. Because 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 to specify which DDP sub-workflow will be executed.

Figure 7 shows the experimental results. As the figure shows, 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 of the input dataset sizes. The results confirm that Kepler facilitates DDP application development with a small execution overhead.

Figure 6. RAMMCP workflow scalability experiment and performance analysis: (a) performance with different available CPU cores, (b) performance with different parallel numbers, and (c) performance with different sequence numbers per execution (log scale).
The reason for the execution time differences for the three DDP combinations is analyzed elsewhere.\(^1\)

Our approach provides 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. Our future work will further study load balancing of DDP patterns and how performance changes based on data location and replication in the Hadoop distributed file system and speculative task launching. We’ll also use 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.

Acknowledgments

We thank the rest of the Kepler and bioKepler teams for their collaboration. This work was supported by bioKepler’s US National Science Foundation ABI Award DBI-1062565.

References


**Jianwu Wang** is the assistant director for research at the Workflows for Data Science (WorDS) Center of Excellence and an assistant project scientist at the San Diego Supercomputer Center at the University of California, San Diego. He’s also an adjunct professor at North China University of Technology, China, and the main researcher for distributed architectures of the open source Kepler Scientific Workflow System. His research interests include Big Data science, scientific workflows, provenance, distributed computing, bioinformatics, observatory systems, conceptual data querying, and software modeling. Altintas has a PhD in computer science from the University of Amsterdam, the Netherlands. Contact her at altintas@sdsc.edu.

**Daniel Crawl** is the assistant director for development at the Workflows for Data Science Center of Excellence at the San Diego Supercomputer Center at the University of California, San Diego. He’s the lead architect for the overall integration of distributed data parallel execution patterns and the Kepler Scientific Workflow System. His research interests include scientific workflows, Big Data, and provenance. Crawl has a PhD in computer science from the University of Colorado, Boulder. Contact him at crawl@sdsc.edu.

**Ilkay Altintas** is the director of the Workflows for Data Science Center of Excellence at the San Diego Supercomputer Center at the University of California, San Diego. She’s a coinitiator of and active contributor to the Kepler Scientific Workflow System. Her research interests include Big Data science, scientific workflows, provenance, distributed computing, bioinformatics, observatory systems, conceptual data querying, and software modeling. Altintas has a PhD in computer science from the University of Amsterdam, the Netherlands. Contact her at altintas@sdsc.edu.

**Weizhong Li** is an associate research scientist at the Center for Research in Biological Systems at the University of California, San Diego. His research focuses on developing computational methods for sequence, genomic, and metagenomic data analysis. Li has a PhD in computational chemistry from Nankai University, China. Contact him at liwz@sdsc.edu.

Selected articles and columns from IEEE Computer Society publications are also available for free at [http://ComputingNow.computer.org](http://ComputingNow.computer.org).

AIP owns AIP Publishing LLC, a scholarly publisher in the physical and related sciences.


*Board members listed in bold are members of the Executive Committee.*

**Management Committee**: H. Frederick Dylla, Executive Director and CEO; Gigi Swartz, Treasurer and CFO; Theresa C. Braun, Vice President, Human Resources; Catherine O’Riordan, Vice President, Physics Resources.