Big Data Analysis using Distributed Actors Framework

Sanjeev Mohindra, Daniel Hook, Andrew Prout, Ai-Hoa Sanh, An Tran, and Charles Yee
MIT Lincoln Laboratory,
244 Wood Street,
Lexington, MA 01810

***Abstract*—The amount of data generated by sensors, machines, and individuals is increasing exponentially. The increasing volume, velocity, and variety of data places a demand not only on the storage and compute resources, but also on the analysts entrusted with the task of exploiting big data. Machine analytics can be used to ease the burdens on the analyst. The analysts often need to compose these machine analytics into workflows to process big data. Different parts of the workflow may need to be run in different geographical locations depending of the availability of data and compute resources. This paper presents a framework for composing and executing big data analytics in batch, streaming, or interactive workflows across the enterprise.**

I. Introduction

The amount of data generated by sensors, machines, and individuals is increasing exponentially. Intelligence, Surveillance, and Reconnaissance (ISR) platforms are moving towards higher resolution sensors and persistence surveillance. This has lead to the enormous volume of data being collected. Similarly, enterprises are collecting large amounts of operational data from Information Technology (IT) systems with the goal of improving operations and cyber security. Finally, the data being generated by people, especially in the context of social media is exploding. With this flow of multi-source data comes the opportunity to extract information in real time that is immediately relevant to users.

Big Data
Decisions

Fig. 1. Goal: Timely, actionable intelligence from big data

II. Computational Workflows

Computational workflows consist of big data analytics working together to produce final data products, not unlike an assembly line producing cars. The logistics for the conversion of raw data to finished data products is governed by the need to most efficiently produce the data products using distributed resources. Just like a manufacturing enterprise, not all data processing happens in one location. And, as in a manufacturing enterprise, it is important to have agile processes so that work can be dynamically shifted from one location to the other. The dynamic routing of work between the various locations provides for a more resilient data enterprise.

Distributed processes can work together to produce and consume data products only if a set of data specifications is agreed upon between the producers and consumers. In addition, automatically generating the input/output libraries for this data from the specification greatly reduces the chances of data incompatibility and decreases time spent on integration.

As shown in Figure 2, both the data and processing capabilities are distributed. The other parallel to the manufacturing enterprise is that the data consumers are also distributed. So a good data communication and distribution model is needed to get the data to the users.

Fig. 2. Challenge: Distributed Data, Computation, and Communication

A. Data Ponds and Data Streams

Every enterprise has several data ponds at different geographic locations, and several sources of data streams. For big data analytics, it is useful to run analytics close to the data and move only the results of the analytics to another location for further analysis. This notion of "computing close to the data" gives rise to distributed workflows. Sometimes, moving data

This work is sponsored by the Department of the Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

from one location to another is unavoidable. If data movement is not done by planes, trains, and trucks, it is useful to run a "quick-look" streaming analytic on the in-flight data stream to provide a heads-up summary of the data. Streaming data analysis is most important for data streams, such as those from sensors and social media where data is naturally generated as a stream and needs to be analyzed in a streaming or online fashion in real-time.

1) Data ponds and data enrichment: The data pond analytics usually operate in an enrichment mode. They take the raw data products from the data store, operate on them to produce other data products, and store these data products back in the data store. These analytics processes are often called Extract, Transform, and Load (ETL) processes. These processes may also keep a history and audit trail of changes for provenance.

2) Data streams and computational pipelines: Streaming data is often processed by analytics in a pipeline. The flow of data through the analytics can be represented as a directed acyclic graph of computational tasks. Each analytic has a well defined set of input data products and output data products. The output streams are buffered using in-memory data structures or persistent data stores.

B. Workflows

Processing of data ponds and streams can result in batch, streaming, and on-demand workflows. Batch workflows are typically used for ETL of large data warehouses. Streaming workflows are typically used for real-time data analysis. On-demand or interactive workflows arise typically when the analyst is querying for data and the query results in analytics being run to produce the data products needed. It provides an interactive model of computing that is often necessary for cued data analysis and data exploration.

C. Workflow Parallelism: Task and data parallel computing

Efficient execution of these workflows presents challenges in both processing and storage. Figure 3 shows an informal architecture diagram at each of the data centers. A single workflow can span several data centers across geographical locations. Previous work [1] has focused on task and data parallelism in a data center to efficiently process computational workflows. In this work, we extend the task parallelism to span across geographical locations.

The data processing and storage infrastructure is typically heterogeneous being composed of parallel and distributed filesystems; SQL and NOSQL databases; virtual machines, GPU clusters, HPC resources, and Hadoop clusters. Not every data center will have support for all of these technologies, but the big data framework should be flexible enough to support any technology in the bottom layer.

The user interface layer can range from graphical user interfaces, interactive Read-Eval-Print-Loop (REPL) environments, GIS environments, etc. The middle layer must deal with heterogeneity of the top and bottom layer. In this paper we mainly focus on the middle layer which involves efficient execution of computational workflows and associated data movement.

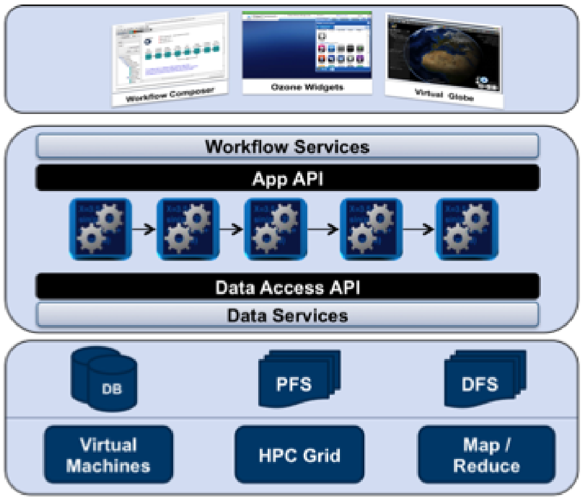

Fig. 3. Notional big-data architecture

III. Challenge Problem

To guide the development of the big data framework, we used the challenge problem as shown in Figure 4. The challenge is to analyze motion imagery, detect moving vehicles, construct tracks, and finally form a graph where the nodes represent locations. Nodes in the graph are joined by an edge if there is a vehicle that travels from one location to the other. The image data is collected at a rate ranging from 100 gigabytes per minute to about 1 terabyte per minute. This processing chain involves image processing, machine learning, and graph analytics making it a good challenge problem for any big data framework. Figure 4 also shows the workflow. Even though the figure depicts the analytics as a linear workflow, the real workflow is a directed acyclic graph.

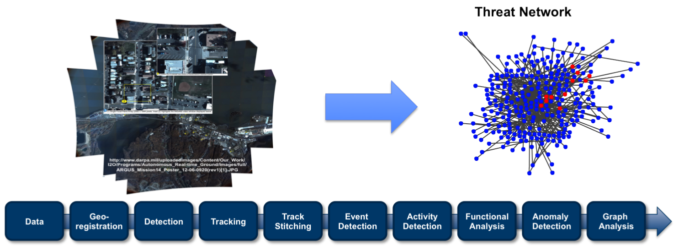

Fig. 4. WAMI challenge problem and workflow

In the following sections, we present the Data Analysis using Distributed Actors (DADA) framework for dealing with big data. The DADA framework is composed of a data architecture and a data processing architecture. The data architecture helps us deal with the data variety and data location challenges, whereas the data processing architecture deals with the challenges of dealing with big data volume, and high data rates on heterogeneous architectures. We start the discussion of the DADA framework with the data architecture, and then move to the processing architecture.

IV. DADA Framework: Data Architecture

The challenge on the data side is driven by the volume, velocity, and variety of data. The data may be stored locally, remotely, or on a distributed file system. The first challenge is to enable analytics to access the data from anywhere. DADA data architecture relies heavily on data services to make this a reality. Data-as-a-Service (DaaS) has two key benefits:

- It provides a universal way to access data through services. For DADA, we rely exclusively on RESTful web services. Using RESTful web services makes it easy to enable a simple polyglot programming model – different analytics in the workflow can be written in different languages. As long as the language allows RESTful web service calls, the analytic can read and write to the data store.
- It allows the analytics to use a simple put/get model to store the domain objects. The actual details of the storage are abstracted away, and may be changed over time without needing a rewrite of the analytics.

The other data challenge stems from the variety of data. There can be variety of data inputs to the workflow. Moreover, a variety of intermediate data products are produced by the various analytics in the workflow. To be able to compose analytics, the output of one analytic, must be readable by its successor in the tool chain. Making this challenge even harder is the fact that the analytics may be written in different programming languages. Furthermore, preference must be given to data formats that are less verbose and minimize network traffic. To solve this problem, we turned to Google Protocol Buffers [2]. Protocol Buffers (protobufs) provide an efficient method for serializing data. The user defines the specifications for the data, and the protobuf compiler generates the code necessary to serialize and deserialize the data in C++, Java, or Python. It allows backward compatibility – the data specification can be modified without breaking existing programs. The key benefit to using protobufs is *compilable data specifications*. This guarantees interoperability between analytics.

Figure 5 shows the key elements of the data architecture.

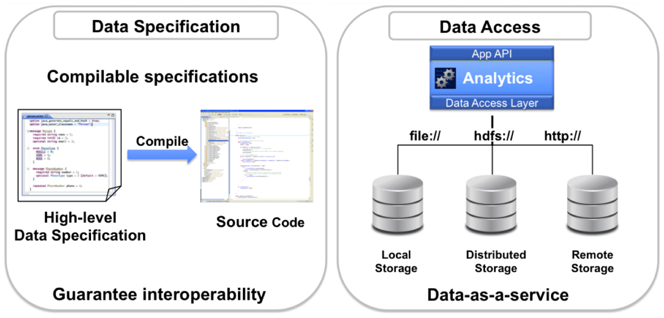

Fig. 5. Data Architecture

In the DADA framework:

- Every data-type persisted on the system has a repository Uniform Resource Identifier (URI) e.g. a directory name or a service name
- Every data persisted on the system has a unique resourceID (for example, a filename)

Taken together, each piece of data stored in the system has a unique URL. The {URL : data} pair provides the Key/Value model for data. This permits the analytics to store and retrieve domain objects as opposed to worrying about database schemas, etc. Furthermore, this permits separation of data and data identities enabling us to broadcast lightweight data objects consisting of data identifiers and other metadata as messages in the system. The receiving party can retrieve the entire data object from the system, if the data object is of interest. We provide two ways of transmitting data between the analytics — using the control plane or the data plane. The data plane is service-oriented DaaS, which provides a simple and robust way of accessing data. For event-driven data flows, we provide a control plane.

Depending on application needs, it is possible to use only the control plane for moving *all* data. However, several use cases require a more persistent, albeit lower performance, data flows and those needs can be met by a combination of data flows on the control plane as well as the data plane. Figure 6 shows the various data flows. In some applications, it is useful to mix-and-match and have lightweight metadata flow through the control plane, and the heavier data objects flow through the data plane.

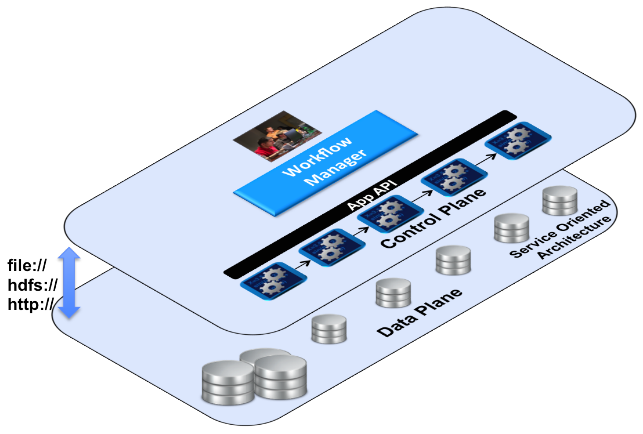

Fig. 6. Data Architecture: Control plane and the Data plane

V. DADA FRAMEWORK: PROCESSING ARCHITECTURE

The key element of any big data processing architecture is that it must have support for both task and data parallelism. For our particular problem domain, the tasks must be able to span several data centers across different geographic locations. We limit our data parallel computing to a single data center, mainly because the fine-grain parallelism of data parallel computations does not lend itself very well to long distance communications.

For task parallelism, we needed a framework that provided a unified local and distributed programming model – from threads to processes to data centers. Increasingly, we need to support heterogeneous platforms (multi-cores, GPUs, function accelerators, etc.) and multiple languages. It is important to have address space isolation guarantees for tasks. Finally, the scalability imperative mandates a framework that can scale resiliently and tolerate faults.

The actor model of concurrent computation satisfied these constraints. It was first proposed by Hewitt in 1973 [3] and is detailed by Agha [4]. It forms the basis of concurrency and parallelism in Erlang and Scala, and there has been a renewed interest because of rise of multi-core architectures [5].

A. Actor Model

The Actor model consists of independent communicating computational agents that share nothing and respond to messages by:

1) Generating more messages
2) Modifying their behavior
3) Creating other actors

Actors communicate with each other using asynchronous communication. Actors are stateful, and modify their behavior in response to messages. The creation of actors on-the-fly is a key element in supporting dynamic scalability and fault tolerance.

B. Extending the actors model for big data analytics

Actors provide a mathematical framework for computational agents. In the DADA framework, they provide a mechanism for incorporating heterogeneous task parallelism. However, actors do not provide a good way of dealing with data parallelism. To exploit data parallelism, DADA provides support for established *non-actor* software stacks such as MPI and Map/Reduce. The data processing framework is composed of the following components:

1) System Actors: System level actors for managing startup, and monitoring health and status of the system
2) User-defined analytic actors: The processes that perform the data processing work
3) Data-Parallel actors: For managing data-parallel computational systems such as MPI and Map/Reduce, or simply command line shell applications.
4) Job Scheduler: Used by system actors for job scheduling and job management
5) Messaging system(s): Move messages between actors
6) Configuration, Naming, and Synchronization services: For managing configuration, and coordinating distributed actors.

For DADA, we chose to adapt existing technologies for our needs. Rapidly evolving actor frameworks such as Akka [5] will provide us most of what we need to craft task parallelism when they mature. “Akka is a toolkit and runtime for building highly-concurrent, distributed, and fault tolerant event-driven applications on the JVM” [5]. Currently, we are not using Akka, but will migrate to using it once it stabilizes and matures. For now we have chosen to shoehorn our actors into bolts in the Storm framework. Storm is a free and open source distributed realtime computation system that makes it easy to reliably process unbounded streams of data [6]. However, Storm suffers from the weakness that workflows, or topologies as they are called, are static. A marriage of Storm and the actor framework, has the potential for providing support for more dynamic workflows with automatic load balancing.

The following sub-sections provide more details on these components.

C. System Actors

System actors are responsible for defining and starting a workflow, including support for data provenance. The system actors are also responsible for monitoring the health and status. The DADA framework treats health and status messages like other messages in the system, and they can either be sent on the control plane, or be persisted using the data plane.

D. Analytic actors

These actors respond to messages, and perform the work. The actors can communicate using the control plane or the data plane.

E. Data-parallel actors

Data-parallel actors provide the bridge to data-parallel computational systems such as MPI and Map/Reduce. The legacy actors themselves live on the control plane and can pass messages to the data plane. These actors launch MPI and Map/Reduce jobs using the job schedulers. The MPI data-parallel jobs do not have direct access to the control plane messages. They can only access and create data plane messages, in addition to their native messaging capability. It is the job of the data parallel actors to bridge the divide.

F. Job Scheduler

The creation of actors on-the-fly is a key element in supporting dynamic scalability and fault tolerance. Creation of new actors involves a) library primitives to spawn jobs and b) job schedulers to satisfy the spawn request. There are a number of schedulers on the market. DADA uses the Grid Engine for job scheduling [7]. The Grid Engine job scheduler is used for launching and scheduling all the MPI, Map/Reduce, and other jobs.

G. Messaging system(s)

The DADA framework provides messaging support for tasks or actors. Systems such as MPI provide support for an orthogonal communication system for point to point or broadcast communications. The DADA framework does not interfere with these data-parallel data flows.

In terms of messaging, actors can deal with input and output of messages in one of two ways: Push-in/ Push-out or Pull-in/Push-out. The push-in / push-out model is efficient way of communication and uses routers to accomplish messaging. Some of the common router topologies are:

- Random router: Route incoming messages randomly to downstream actors
- Round-robin router: Route incoming messages to downstream actors in a round-robin manner
- Broadcast router: Broadcast incoming messages to all downstream actors
- Priority router: Prioritize the incoming messages for delivery
- Filtered router: Filter the incoming messages for delivery

Another method of communication is for the actors and computational agents to push the message to a buffered queue; and for the receiving actor to pull it off the queue. Some of the common queue topologies are:

- Point-to-point: Single-producer, single-consumer queue
- Work queues: Single-producer, multiple-consumer queue
- Publish/Subscribe: Multiple-publisher, multiple-subscriber
- Priority queues: Priority queue for messages

The DADA framework provides support for both these models of communication so that the users of the framework can select how to do their messaging based on their needs. Typically, the control plane messaging is done using the Push-in/Push-out model because it provides a low latency messaging system. Similarly, the Push-in/Pull-out mode of communication is typically easier to use and can be implemented using message queues. The Push-in/Pull-out model may even be implemented using simple data services for batch workflows, or if the ordering of messages is unimportant.

H. Configuration, Naming, and Synchronization services

For very simple workflows, it is sufficient to have a centralized filesystem-based configuration and coordination system. However, as complexity grows, it is essential to provide a more robust set of capabilities. Like many other distributed processes, DADA uses Apache ZooKeeper [8] to allow distributed actors and processes to coordinate with each other.

VI. Conclusion

The DADA framework allows enterprises to deal with their data ponds and data streams by composing batch, streaming, or interactive workflows. A single workflow can span several geographical locations. The DADA framework is well suited to heterogeneous infrastructures for computing and storage; and is comprised of a data architecture and a data processing architecture. The data architecture addresses the data variety and data location challenges, whereas the data processing architecture addresses the challenges of dealing with big data volume and high data rates on heterogeneous architectures.

To ensure composability of analytics, the DADA framework uses Google protocol buffers (protobufs) to serialize/deserialize data. The protobuf compiler generates the infrastructure necessary to serialize/deserialize the data in many different languages and this provides us with *compilable data specifications*, which guarantee interoperability between analytics. Every piece of data persisted on the system has a unique URL, and this provides a simple model for analytics to deal with data.

The DADA framework provides two ways of transmitting data between the analytics — using the control plane or the data plane. The data plane is service-oriented DaaS, which provides a simple and robust way of accessing data. For event-driven data flows, we provide a control plane. Depending on application needs, it is possible to use only the control plane for moving *all* the data.

DADA provides support for both task and data parallelism. The actor model provides DADA with a solid theoretical framework for unified local and distributed programming – from threads to processes to data centers. The ability to provide support for MPI-style data parallelism sets DADA apart from some of the other frameworks which are essentially only task parallel approaches to big data.

We have implemented the framework, using a real challenge problem involving image processing, machine learning, and graph analytics as a forcing function. The framework has been developed by adapting open-source technologies, and we plan to release our work as open source as well.

Acknowledgment

The authors would like to thank the ASD R&E, OSD, and ONR for their support of this effort under the Data to Decisions program. We would also like to thank our partners on this program from AFRL, ARL, and SPAWAR SSC.

References

[1] Sanjeev Mohindra et. al. Task and conduit framework for multicore systems, *DoD HPCMP Users Group Conference, 2008.* pp. 506–513.

[2] Protocol Buffers http://code.google.com/p/protobuf

[3] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[4] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[5] Akka http://akka.io/

[6] Storm http://storm-project.net/

[7] GridEngine http://www.sun.com/grid/.

[8] Hunt, P. and Konar, M. and Junqueira, F.P. and Reed, B., ZooKeeper: Wait-free coordination for Internet-scale systems, Proceedings of the 2010 USENIX conference on USENIX annual technical conference, 2010