Bringing the HPC reconstruction algorithms to Big Data platforms*

Nikolay Malitsky
NSLS-II Department
Brookhaven National Laboratory
P.O. Box 5000, Upton, NY 11973, USA
malitsky@bnl.gov

***Abstract*—The HPC and Big Data technologies are two important modern means of the scientific discovery process, but until recently, their development was evolved along two independent paths. Born and matured prior to the Big Data era, HPC scientific-oriented applications were implemented with the Message Passing Interface (MPI) protocol. MPI, however, was not designed for data-intensive applications and therefore requires project-specific extensions. An efficient processing framework offered by the Spark computational platform resolved many issues and built a strong foundation for their subsequent integration. At this time, the Spark ecosystem encompasses a rich collection of data analysis algorithms covering everything from SQL queries, to machine learning, to graph processing. Therefore, the paper proposes to extend this scope with the HPC applications of light source experimental facilities. Specifically, it focuses on the Spark-based integration of the SHARP distributed ptychographic solver developed by the team of the Center for Advanced Mathematics for Research Applications (CAMERA) at Berkeley. Aside from its practical value, this application represents an excellent reference use case that captures the major technical aspects of other beamline applications. The resulting approach is derived from the analysis of alternative communication models implemented in Spark-based distributed deep learning frameworks.**

Keywords—Big Data, HPC, Spark, MPI, experimental facility

I. Introduction

Advances in detectors and computational technologies provide new opportunities for applied research and the fundamental sciences. Concurrently, dramatic increases in the three V's (Volume, Velocity, and Variety) of experimental data and the scale of computational tasks produces demands for new data management and processing systems of experimental facilities. Historically, the development of these systems was associated with the operation and analysis of the High Energy and Nuclear Physics experiments led by large international collaborations. Recently, a list of large-scale data-intensive projects was significantly extended with other science drivers, such as telescopes, fusion reactors, and light sources [1-5]. For example, NSLS-II [6] represents the latest synchrotron facility, bringing advances of experimental technologies to a wide range of disciplines; from condensed matter physics to biology. One of the major breakthroughs is associated with the modern x-ray detectors capable of producing multi-megapixel images with kilohertz rates. The operation of corresponding beamlines and short-term projections immediately highlighted the importance of real-time and near-real-time data processing and reconstruction pipelines aiming to automate and accelerate a pathway for scientific discovery.

The demand for near-real-time processing pipelines of light source experiments is already gradually addressed by other light source facilities using existing workflow management systems, such as Spot [7] and Swift/T [8]. In contrast with conventional batch applications, streaming pipelines require the coordination and interconnection of multiple heterogeneous high performance computing (HPC) tasks. For example, the micro-tomography reconstruction pipeline of the ALS Beamline 8.3.2 [7] can contain up to 30 unique steps: normalization, sonogram generation, reconstruction, HDF5 packaging, and others. Most of these steps are highly parallel and processed as MPI jobs utilizing different numbers of workers. Therefore, to glue them together, the Spot workflow management system adds additional components, such as a workflow engine, reliable message queues, and a worker farm. In Swift/T, the same application can be handled directly within the MPI framework using the Turbine dataflow engine and the Asynchronous Dynamic Load Balancer (ADLB) library. The general approach for the integration of the HPC and stream models has also been considered in several MPI-based research projects. For example, Emilio Mancini and colleagues [9] proposed a hybrid model mixing intra and inter cluster communications. Another team [10] developed a stream library on the top of MPI bringing new model components, such as channel and stream, and associated operations. At this time, these and other similar research projects, however, represent a collection of disjoined approaches and require substantial effort for their consolidation. Therefore, this paper explores an opposite approach by extending the scope of the Big Data integrated platforms with scientific algorithms originally implemented within the MPI model.

In contrast to the HPC scientific-oriented applications, a large category of data-intensive processing algorithms can be expressed within an embarrassingly parallel model captured by the MapReduce framework. Addressing immediate data-intensive applications, the MapReduce computing platform played a significant role in the expansion of the Apache ecosystem and the formation of a reference level for developing new technologies. This level has been further elevated by the

This research was funded by the National Synchrotron Light Source II, a U.S. Department of Energy (DOE) Office of Science User Facility operated for the DOE Office of Science by Brookhaven National Laboratory under Contract No. DE-SC0012704.

Spark programming model [11] based on resilient distributed datasets (RDDs) and a comprehensive collection of RDD transformations and actions. The new model of the Spark computing platform significantly advanced and extended the scope of data-intensive applications, spreading from SQL queries to machine learning to graph processing.

The advancement of data-intensive technologies has been going along with the consistent extension and consolidation of the Big Data landscape within all its dimensions. One of the most influential solutions of this direction was provided by the lambda architecture [12] bringing together batch and streaming platforms. Spark facilitated this integration by adding the Spark Streaming module [13] implementing a micro-batch processing pattern. Technically, this module reused and extended the RDD-based batch processing framework with the new programming abstraction called a discretized stream, a sequence of RDDs, processed by micro-batch jobs. Adherence to the RDD model automatically provided the Spark streaming applications with the same functional interface and strong fault-tolerance guarantees, including exactly-once semantics. Recently, the integration of batch and streaming platforms was further extended and generalized within the Google Dataflow model [14] that considered batch, micro-batch, and streaming data processing pipelines from the perspective of one unified model structured around four major question-dimensions: What, Where, When and How. Within the Apache stack, this model is developed as the Apache Beam API [15] that can be executed on top of different runners including Spark and other engines.

The combination of a data-intensive processing framework with a consolidated collection of diverse data analysis algorithms offered by Spark represents a strong asset for its application in large-scale scientific facilities across different phases of the knowledge discovery path. Therefore, this paper proposes to challenge the Spark approach and extend its scope with the implementation of the HPC beamline applications as shown in Fig. 1. Specifically, the project focuses on the Spark-based integration of the SHARP ptychographic solver [16] developed by the team of the Center for Advanced Mathematics for Research Applications (CAMERA) at Berkeley. Aside from its practical value, this application represents an excellent reference use case capturing major technical aspects of other beamline applications, for example, tomographic reconstruction [17] and GISAXS simulation [18].

The reminder of the paper is organized as follows. Section 2 provides a brief overview of the Spark platform in the context of data-driven tasks of experimental facilities. Section 3 describes the SHARP ptychographic solver and its multi-GPU parallel model. Then, Section 4 overviews three different parallel models of Spark-based distributed deep learning solvers. This overview is followed by Section 5 presenting the SHARP-SPARK approach merging solutions of the previous section. Finally, Section 6 and 7 discuss related work and concluding thoughts, respectively.

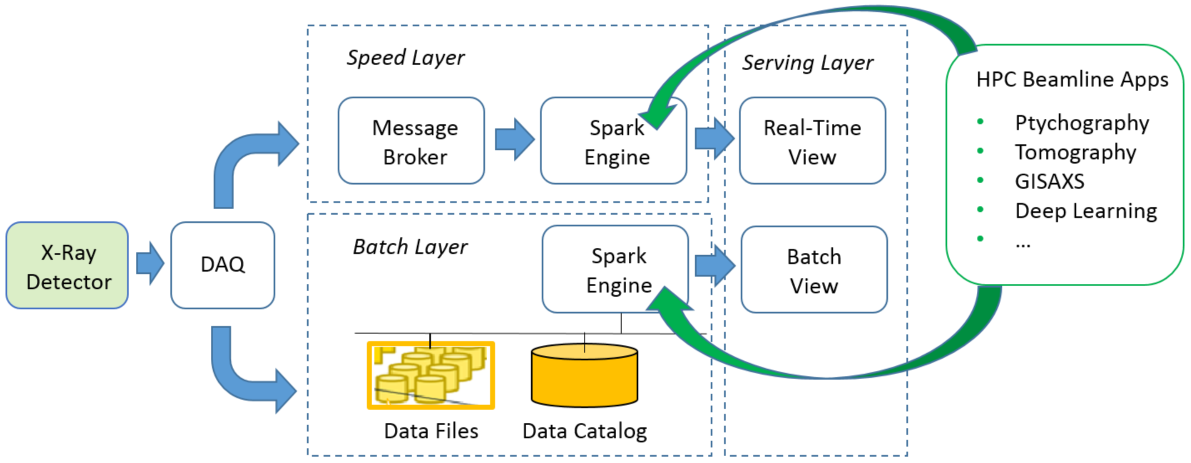


Fig. 1. Spark-based lambda-oriented architecture in the context of beamline applications

II. Spark

Spark [11] is a cluster computing platform introducing a new programming model for efficient and interactive processing of large-scale datasets. It is designed to cover a wide range of workloads that previously required separate distributed systems, including batch applications, iterative algorithms, interactive queries, and streaming. This combination forms a powerful processing ecosystem for building data analysis pipelines and supporting multiple higher-level components specialized for various workloads.

Fig. 2 shows a general overview of the Spark components and proposed extensions driven by tasks of experimental facilities. Spark Core contains the basic functionality of Spark, including components for task scheduling, memory management, interaction with storage systems, and others. Under the hood, Spark is design to efficiently scale up from one to many thousands of compute nodes. To achieve this, Spark can run over a variety of cluster managers, including Hadoop YARN, Apache Mesos and internal Standalone Scheduler. The top layer is represented by an open collection of high-level components addressing different types of data models and processing algorithms.

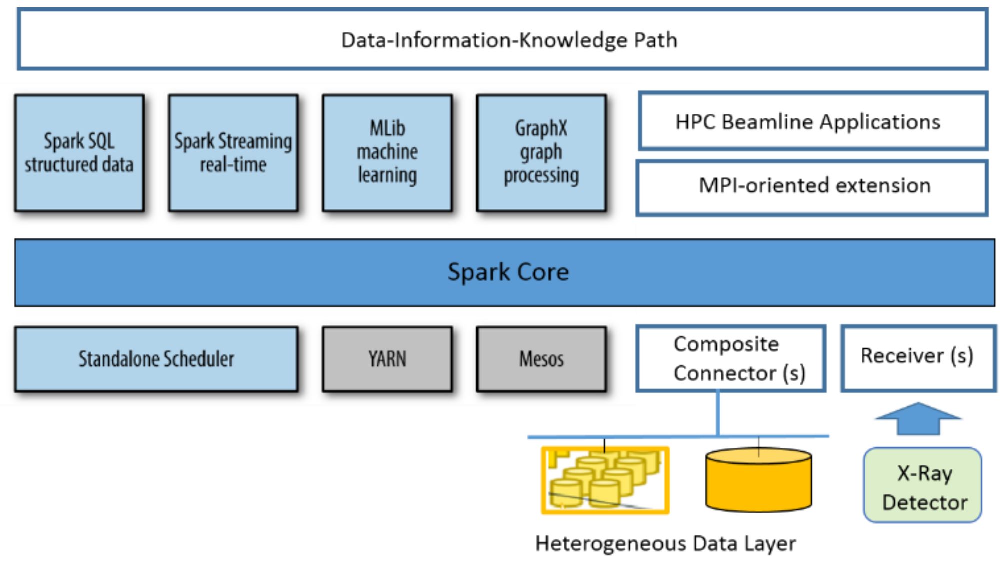


Fig. 2. Spark ecosystem and proposed extensions for experimental facilities

The Spark framework introduces a new abstraction called resilient distributed datasets (RDDs), which are fault-tolerant collections of in-memory containers. The RDD objects can be processed in parallel using a rich set of coarse-grained transformations. In addition, they can be initialized from multiple data sources, including the Hadoop distributed file system (HDFS) and relational and NoSQL databases. Initialization of the Spark resilient distributed datasets is encapsulated in the SparkContext interface and explicitly separated from the processing framework. As a result, RDDs serve as a large-scale generic front-end of heterogeneous distributed data sources.

In contrast with existing data management and analytics systems, this in-situ approach does not require the transformation of data into different formats and provides a natural solution for the connection of the Spark processing framework with composite data layers of experimental facilities. For example, the NSLS-II data layer represents a distributed collection of heterogeneous data sources including multiple components, such as the meta-data store and repositories of data files with time series of control data, detector images, and post-processing results from analysis and reconstruction applications. In Spark, the corresponding composite interface can be implemented with multi-stage pipeline moving intra-store relationship aspects to the RDD middle layer.

The application of the Spark platform to experimental facilities, however, requires additional studies associated with the deployment and development of scientific-oriented algorithms used across different phases of the knowledge discovery path. The current version of the Spark framework is implemented in the Scala and Java languages. On the other hand, most scientific applications are being developed with the C++ ecosystem including the Fortran and Python bindings. Therefore, these applications are supported by the additional inter-language adapter, PySpark, shown in Fig 3. On the client side, PySpark launches a JVM, serializes and sends the Python functions to distributed Spark workers. In turn, remote workers launch Python processes and establish communication with a client driver program. The communication framework is deliberately designed. For example, large data transfers are performed directly between the Python programs without the Java overhead. In addition, to address new requirements, it offers the pluggable mechanism for supporting different serialization approaches and communication protocols.

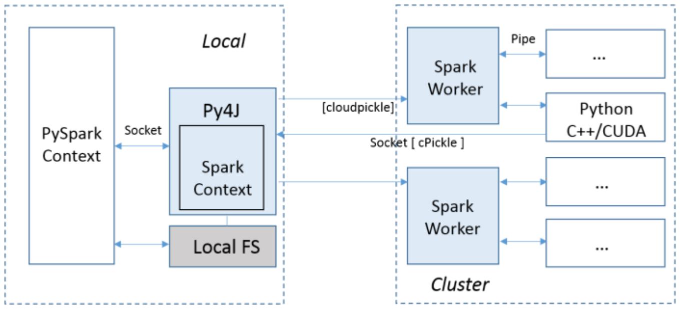


Fig. 3. PySpark [19]

The PySpark approach provides an immediate solution for integrating a new category of C++ and GPU applications including distributed versions of deep learning frameworks, such as Caffe [20] and TensorFlow [21]. This paper follows this direction and aims to explore and advance it in the context of the HPC reconstruction algorithms of light source experiments.

III. Ptychographic Application

Ptychography is one of essential image reconstruction techniques used in light source facilities. It was originally proposed for electron microscopy [22] and lately applied to X-ray imaging [23-24]. The method consists of measuring multiple diffraction patterns by scanning a finite illumination (also called the probe) on an extended specimen (the object). Then, the redundant information encoded in overlapped illuminated regions is used for the reconstruction of the sample transmission function. Specifically, under the Born and paraxial approximations, the measured diffraction pattern for the *j*th scan position can be expressed as:

$$I_j(\mathbf{q}) = |\mathbf{F}\psi_j|^2 , \tag{1}$$

where **F** denotes Fourier transformation, **q** is a reciprocal space coordinate, and ψ_j represents the wave at the exit of the object O illuminated by the probe P:

$$\psi_j = P(\mathbf{r}-\mathbf{r}_j)O(\mathbf{r}) \tag{2}$$

Then, the object and probe functions can be computed from the minimization of the distance $\|\Psi - \Psi^0\|^2$ as [25]:

$$\varepsilon = \|\Psi - \Psi^0\|^2 = \sum_j \sum_r |\psi_j(\mathbf{r}) - P^0(\mathbf{r} - \mathbf{r}_j)O^0(\mathbf{r})|^2 \tag{3}$$

$$\frac{\partial\varepsilon}{\partial P^0} = 0:\ P^0(\mathbf{r}) = \frac{\sum_j \psi_j(\mathbf{r}+\mathbf{r}_j)O^*(\mathbf{r}+\mathbf{r}_j)}{\sum_j |O(\mathbf{r}+\mathbf{r}_j)|^2} \tag{4}$$

$$\frac{\partial\varepsilon}{\partial O^0} = 0:\ O^0(\mathbf{r}) = \frac{\sum_j \psi_j(\mathbf{r})P^*(\mathbf{r}-\mathbf{r}_j)}{\sum_j |P(\mathbf{r}-\mathbf{r}_j)|^2} \tag{5}$$

These minimization conditions need to be augmented with the modulus constraint (1) and be included in the iteration loop. For example, the comprehensive overview of different iterative algorithms is provided by Klaus Giewekemeyr [26]. At this time, the difference map [27] is considered as one of the most generic and efficient approach addressing these types of imaging problems. It finds a solution in the intersection of two constraint sets using the difference of corresponding projection operators, π_1 and π_2, composed with associated maps, f_1 and f_2:

$$\begin{gathered}\psi^{n+1} = \psi^n + \beta\,\Delta(\psi^n)\\ \Delta = \pi_1 \circ f_2 - \pi_2 \circ f_1\\ f_i(\psi) = (1+\gamma_i)\pi_i(\psi) - \gamma_i\psi\end{gathered} \tag{6}$$

where $\gamma_{1,2}$ are relaxation parameters. In the context of the ptychographic applications, these projection operators are associated with the modulus (1) and overlap (2) constraints as:

$$\pi_a(\psi): \psi \rightarrow \mathbf{F}^* \frac{F\psi}{|F\psi|}\sqrt{I} \tag{7}$$

$$\pi_o(\psi): \psi \rightarrow P(\mathbf{r}-\mathbf{r}_j)O(\mathbf{r}) \tag{8}$$

For example, by selecting different values of relaxation parameters, the difference map (6) can be specialized to different variants of phase retrieval methods, such as the hybrid input-output (HIO [28]) :

$$\psi^{n+1} = [2\pi_a\pi_o - \pi_a + (1 - \pi_o)]\psi^n \tag{9}$$

and hybrid projection-reflection (HPR [29]) algorithms. By further developing HPR, Russel Luke [30] introduced the relaxed averaged alternating reflections (RAAR) approach:

$$\psi^{n+1} = [2\beta\,\pi_o\pi_a + (1 - 2\beta)\pi_a + \beta(1 - \pi_o)]\psi^n \tag{10}$$

The RAAR algorithm was implemented in the SHARP program [16] at the Center for Advanced Mathematics for Research Applications (CAMERA). SHARP is a high-performance distributed ptychographic solver using GPU kernels and the MPI protocol. As shown above, most equations with the exception of (4) and (5) are framewise intrinsically independent. Therefore, the ptychographic application is naturally parallelized by dividing a set of data frames among multiple GPUs. Then, for updating a probe and an obejct, the partial summations of (4) and (5) are combined across distributed nodes with the MPI protocol.

In addition to the high-performance approach, the SHARP software suite provides a consistent object-oriented framework for integrating other algorithms and adding new extensions, such as partially coherent illumination [31], fly-scan ptychography [32], and probe reconstruction with not well-defined initial conditions. This feature is especially important for the development and commissioning of new beamline experiments. For example, Fig. 4 shows an off-centered probe representing a substantial challenge to conventional algorithms.

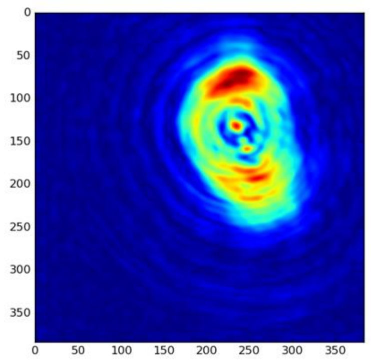


Fig. 4. Probe amplitudes

As a result, SHARP was successfully applied to the Hard X-Ray Nanoprobe (HXN) beamline at the NSLS-II facility by integrating several extensions of the HXN Python program. Table I includes profiling results produced by the SHARP-NSLS2 package for reconstructing an object shown in Fig 5.

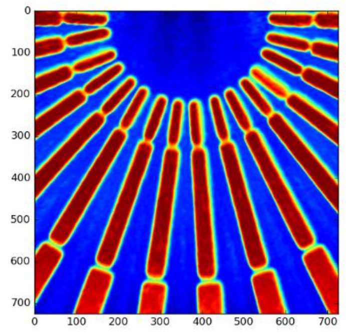


Fig. 5. Object phases

SHARP-NSLS2 significantly boosted the performance of the off-line data processing and immediately highlighted the topic associated with its run-time online applications. Moreover, the integration of the SHARP single-GPU solver within the Spark streaming platform does not require any additional extensions and can be naturally handled by PySpark through the SHARP Python interface.

TABLE I. PROFILING RESULTS OF THE SHARP-NSLS2 ITERATION STEP PROCESSED ON A SINGLE GPU (TESLA K20)

Function and Equations	**Time, s**	
	256 frames	***512 frames***
HIO method (9) including the modulus (7) and overlap (8) projections	0.06	0.12
Probe update (4)	0.025	0.05
Object update (5)	0.03	0.06
	0.115	0.23

The MPI-based distributed version of the SHARP solver (see Fig. 6) however introduces a problem associated with the implementation of the inter-engine communication for updating a probe and an object according to (4) and (5). In SHARP, it is based on the MPI Allreduce method.

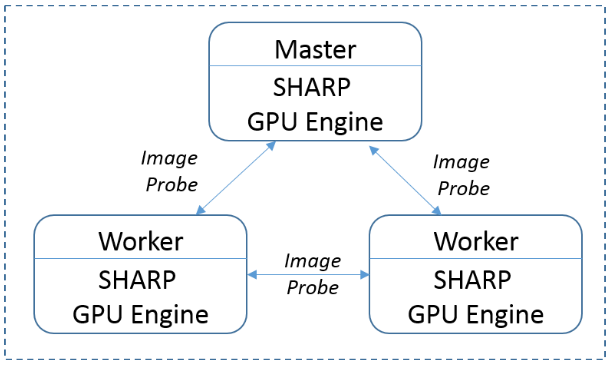


Fig. 6. MPI communication model of the SHARP distributed solver

As mentioned above, this topic is not explicitly addressed by the Spark model. But because of its importance, it cannot be avoided and has been recently considered in the context of the emerging deep learning applications.

IV. SPARK-BASED DISTRIBUTED DEEP LEARNING SOLVERS

Deep learning is an active area of machine learning, influencing significant improvements in a number of application domains over the last few years. These models are based on heterogeneous multi-layer networks that are usually trained with variants of the stochastic gradient descent (SGD) algorithm. Many of recent advances involved the training of large networks and required substantial computational resources [33]. One of the major challenges of these large-scale machine learning systems is associated with the parallelization of the SGD optimization algorithm. Within the Spark computing platform, the development of the distributed deep learning solvers was addressed by several projects, such as SparkNet [34], CaffeOnSpark [35], and TensorSpark [36].

SparkNet directly relied on the Spark driver-executor scheme consisting of a single driver and multiple executors (see Fig. 7) running the Caffe or TensorFlow deep learning solvers on its own subset of data. In this approach, a driver communicates with executors for aggregating gradients of model parameters and broadcasting averaging weights back for subsequent iterations.

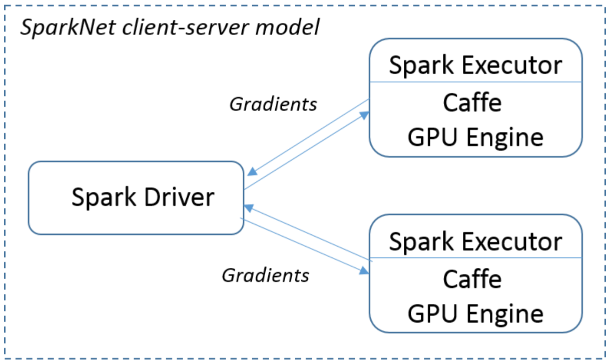


Fig. 7. SparkNet client-server model

According to the SparkNet-based benchmark, the driver-executor scheme introduces a substantial communication overhead that can be minimized by subdividing the optimization loop into chunks of iterations. Addressing the same problem, Yahoo's team proposed a peer-to-peer variant (see Fig. 8) providing a MPI Allreduce style interface over Ethernet or Infiniband. The corresponding approach is implemented within the CaffeOnSpark project [35].

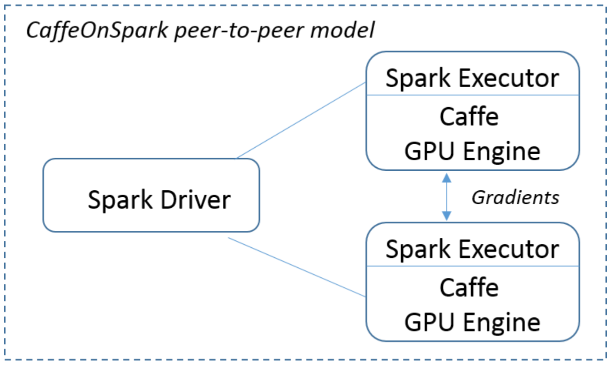


Fig. 8. CaffeOnSpark peer-to-peer model

CaffeOnSpark is a composite package encompassing multiple components written in different languages. The communication among C++ Caffe engines is implemented with a compact C++ library supporting two protocols and different approaches for accessing CPU and GPU memories. The Driver and Worker components, including the SGD optimization loop and coordination of distributed Workers, is written in Java and Scala. Moreover, it represents a substantial part of the CaffeOnSpark package and requires the corresponding development effort for applying this method to different types of processing engines. This issue is eliminated within the TensorSpark lightweight approach based on the PySpark middle layer and the Tornado server (see Fig. 9).

Similar to SparkNet and CaffeOnSpark, TensorSpark's Driver implements the SGD optimization loop and coordinates distributed engines via the Spark interface including its ability to cache and keep data in memory on distributed workers between operations. In contrast with other approaches, TensorSpark uses the Tornado Python web server to exchange model parameters among engines. The script fragment of the Driver program in Fig. 10 outlines this approach.

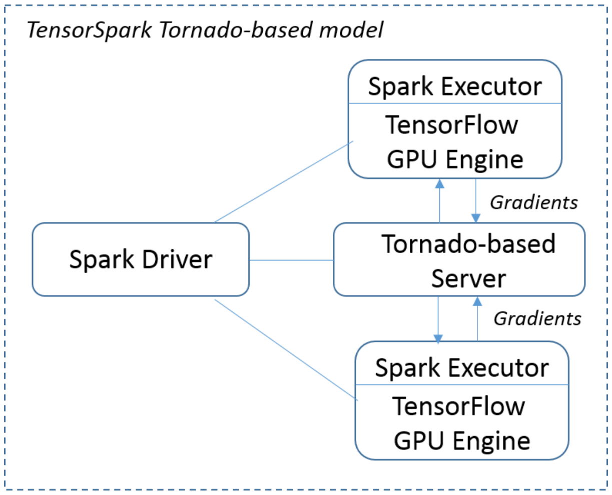
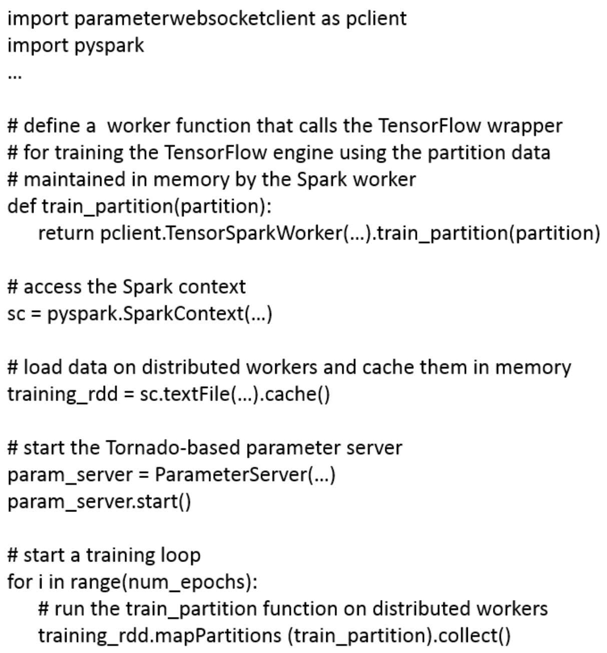


Fig. 9. TensorSpark Tornado-based model

Following the Spark programming model, the Driver program begins by accessing the Spark context and creating the RDD datasets for applying subsequent operations, transformations and actions. In this script, the RDD datasets are created by loading external data files. In addition, these datasets are cached in memory on distributed workers. In the end of the initialization step, the program starts the Tornado-based server and then moves to the optimization part.



```
import parameterwebsocketclient as pclient
import pyspark
...

# define a  worker function that calls the TensorFlow wrapper
# for training the TensorFlow engine using the partition data
# maintained in memory by the Spark worker
def train_partition(partition):
    return pclient.TensorSparkWorker(...).train_partition(partition)

# access the Spark context
sc = pyspark.SparkContext(...)

# load data on distributed workers and cache them in memory
training_rdd = sc.textFile(...).cache()

# start the Tornado-based parameter server
param_server = ParameterServer(...)
param_server.start()

# start a training loop
for i in range(num_epochs):
    # run the train_partition function on distributed workers
    training_rdd.mapPartitions (train_partition).collect()
```

Fig. 10. Example of the TensorSpark script fragment

Each iteration of the optimization loop applies the mapPartitions transformation of the RDD dataset for shipping the train_partition function to distributed workers and running it with cached data of the associated dataset. The function calls the TensorFlow wrapper that calculates gradients and updates model parameters via the Tornado-based server.

From the general perspective, all three approaches represent alternative solutions addressing one common task: implementation of the Spark inter-worker communication across a sequence of the RDD operations. The current version of the Spark communication model provides only a few related mechanisms, such as accumulators and broadcast variables, which do not address requirements of considered applications. On the other hand, as shown by corresponding projects, the Spark framework is open for new extensions.

V. SHARP-SPARK APPROACH

The projects described in the previous section extended the scale of existing GPU-based deep learning solvers. As a result, the corresponding distributed versions augmented the original source code with parallel components. In contrast with these applications, the HPC scientific-oriented algorithms are already parallelized using the MPI protocol. Then, an ideal approach for their integration with the Spark ecosystem would be the implementation of the Spark MPI-oriented extension. This task, however, requires a substantial research and development effort. On the other hand, there is a large category of the HPC applications relying on a few MPI methods. Therefore the paper suggests to explore this direction in the context of leading reconstruction applications, such as the SHARP ptychographic solver. Fig. 11 shows the UML diagram of the corresponding SHARP-SPARK approach.

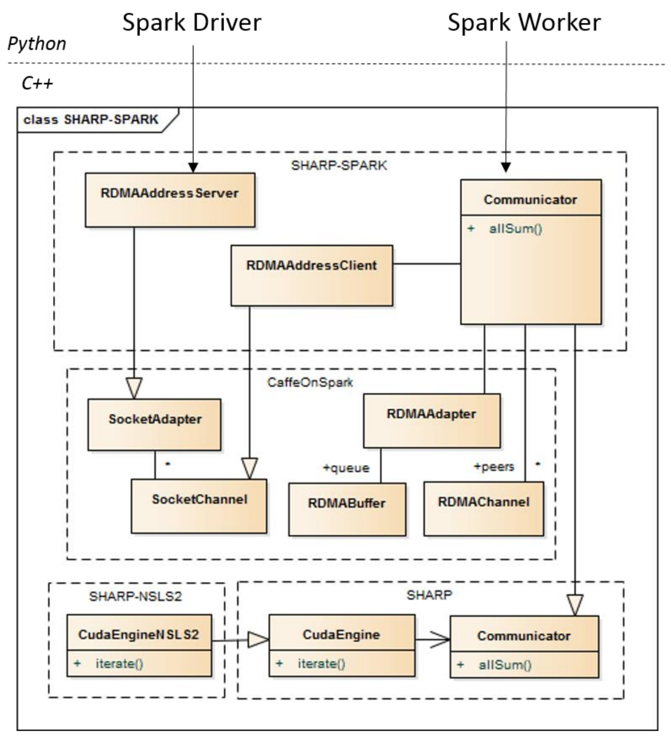


Fig. 11. Structure of the SHARP-SPARK approach

The SHARP-SPARK package follows the CaffeOnSpark RDMA peer-to-peer model and augments it with the RDMA address exchange server based on the CaffeOnSpark socket library. The SHARP framework significantly facilitates this implementation by encapsulating the inter-engine communication in the Communicator class. According to this approach, the Driver program starts the address exchange server and sends its address to distributed workers as an argument of the RDD function (see Fig. 12). In the beginning of this function, a worker creates a Communicator instance that establishes RDMA peer-to-peer channels, pre-allocate memory buffers and exchanges its RDMA addresses (local identifier, queue pair number and packet sequence number) within the *connect* method.

```
from pyspark import SparkContext
import sharpspark
...
sc = SparkContext(appName="SharpSpark")

partitions = 4

srv = sharpspark.AddressServer.createServer()
addr = srv.start(partitions)

inputs = []
for p in range(0, partitions):
   args = {
      'addr' : addr,
      'rank' : p,
      'size'  : partitions,
   }
   inputs.append(args)

# define worker's function for running a parallel solver
def wf(args):
   ...
   comm = sharpspark.Communicator.createCommunicator(args['rank'], args['size'])
   comm.allocate(imageSize)
   comm.connect(args['addr'])
   ...

# Process a parallel solver on distributed workers
outs = sc.parallelize(inputs, partitions).map(wf).collect()
```

Fig. 12. Fragment of the SHARP-SPARK driver script

```
def f(args):

   comm = sharpspark.Communicator.createCommunicator(args['rank'], args['size'])

  # allocate buffers used in the peer-to-peer communication
   imageSize = 2*1000000
   comm.allocate(imageSize*4)

   # connect to the address server and exchange the RDMA addresses
   comm.connect(args['addr'])

   # define a local array (e.g. image)
   a = np.zeros(imageSize, dtype=np.float32)
   a[imageSize-1] = 1.0

  # sum peers' arrays for several iterations
   t1 = datetime.now()
   for i in range(0, 10):
      comm.allSum(a)
   t2 = datetime.now()

  # prepare and return the benchmark results
   out = {
      'a' : a[imageSize-1],
      'time' : (t2-t1),
   }

   comm.release()

   return out
```

Fig. 13. Worker's function of the benchmark application

In contrast with the CaffeOnSpark approach, the address exchange server does not rely on the Spark-based caching mechanism and, as a result, the Java intermediate layer. Moreover, the Communicator class is application-neutral and can be used independently with other related engines. For example, Fig 13 shows worker's function used for benchmarking this approach. It sums large arrays of floats across the Spark workers and returns timing results to a driver program. The corresponding application was deployed on the local cluster and demonstrated comparable performance relative to the MPI version (see Table II).

TABLE II. PROFILING RESULTS OF THE MPI-BASED AND SHARP-SPARK APPROACHES PRODUCED ON A CLUSTER WITH 4 GPU NODES

Approach	Time, s
MPI Allreduce based on the MVAPICH2 implementation [37]	0.013
SHARP-SPARK based on the CaffeOnSpark communication library [35]	0.016

VI. RELATED WORK

One of the most comprehensive overviews of the Big Data and HPC ecosystems is compiled by G. Fox and colleagues. Their application analysis [38] encompasses several surveys, such the NIST Big Data Public Working Group and NRC reports, including multiple application domains: energy, astronomy and physics, climate, and others. Another paper of this team [39] provides the comparison of the Apache Big Data Stack (ABDS) and HPC technologies using a list of 289 software projects. This analysis explicitly highlights common and complementary approaches of two ecosystems through 21 layers of a reference HPC-ABDS architecture. The convergence of the Big Data and HPC ecosystems is also addressed by several recent DOE workshops [3][5][40].

Development of the Spark applications on HPC systems is becoming an active topic. For example, in 2015, Cray and researchers at UC Berkeley's AMPLab and NERSC facility established a collaboration to advance the Spark platform on HPC systems. Recently, this direction was covered in several papers. First [41], the NERSC team together with researches from University of Oregon evaluated Spark on the Edison and Cori Cray XC supercomputers with BiData Benchmark, GroupBy and PageRank. Their results demonstrated scalability up to O(104) cores. Another paper [42] of the NERSC team presented a parallel I/O API, called H5Spark, for accessing scientific data stored in the HDF5 and NetCDF file formats on the Lustre parallel file system. Two other papers [43-44] demonstrated the importance of the extension of the Spark model with MPI-like collective communications addressed by the SHARP-SPARK application.

VII. CONCLUSIONS

The paper proposes the implementation of the HPC reconstruction and simulation algorithms within the Spark batch and streaming platforms for addressing Big Data challenges faced by light source experimental facilities. Spark is an emerging technology accumulating and consolidating technical solutions required by different phases of the knowledge discovery path of scientific experiments: in-situ data access of heterogeneous data sources, support of micro-batch processing pipelines, and an open collection of data analysis algorithms ranging from SQL queries to machine learning to graph processing. The current version of the Spark ecosystem however does not encompass the HPC applications and the paper demonstrates this direction in the context of the Spark-based extension of the SHARP MPI/GPU ptychographic solver. The implemented approach is derived from the analysis and composition of several techniques introduced in Spark-based distributed deep learning frameworks. The resulting application validates this direction and establishes the guidance for the development of the MPI-oriented extension.

Acknowledgment

The development of the SHARP-SPARK application has benefitted from the contributions and suggestions of several teams: researchers and developers of the SHARP program, the CaffeOnSpark team, the HXN and control groups of the NSLS-II project, and the Computational Science Laboratory of the BNL Computational Science Initiative. Particularly, we would like to thank Nicholas D'Imperio, Matt Cowan, Andrew Feng, Leon Flaks, Annie Heroux, Xiaojing Huang, Harinarayan Krishnan, Kenneth Lauer, Stefano Marchesini, Talita Perciano, Rafael Perez, Robert Petkus, James Sethian, David Shapiro, and Dong Zhihua.

References

[1] K. K. Van Dam et al., Chalenges in Data Intensive Analysis at Scientific Experimental User Facilities, Handbook of Data Intensive Computing, Springer, 2011.

[2] P. Nugent et al., "Data and Communications in Basic Energy Sciences: Creating a Pathway for Scientific Discovery," DOE Workshop Report, 2012.

[3] V. Sarkar et al., "Synergistic Challenges in Data-Intensive Science and Exascale Computing," DOE ASCAC Data Subcommittee Report, 2013

[4] Working Group. Accelerating Scientific Knowledge Discovery, DOE Workshop Report, 2013

[5] M. Berry et al., "Machine Learning and Understanding for Intelligent Extreme Scale Scientific Computing and Discovery," DOE Workshop Report, 2015.

[6] Q. Shen, S. Dierker, and J. Hill, "NSLS-II Strategic Plan," BNL, 2015.

[7] J. Deslippe et al., "Workflow Management for Real-Time Analysis of Light Source Experiments," Workshop on Workflows in Support of Large-Scale Science, 2014.

[8] J. Wozniak et al., "Swift/T: Scalable data flow programming for distributed-memory task-parallel applications, " CCGrid, 2013.

[9] E. Mancini, G. March, and D. Panda, "An MPI-Stream Hybrid Programming Model for Computational Clusters," CCGrid, 2010.

[10] I. Peng et al., "A Data Streaming Model in MPI," ExaMPI, 2015.

[11] M. Zaharia et al., "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," NSDI, 2012.

[12] N. Marz and J. Warren, Big Data: Principles and Best Practices of Scalable Realtime Data Systems, Manning Publications, 2015.

[13] M. Zaharia et al., Descretized Streams: Fault-Tolerant Streaming Computation at Scal"e, SOSP, 2013.

[14] T. Akidau et al., The Data Flow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing," VLDB, 2015.

[15] Apache Beam (incubating), URL http://beam.incubator.apache.org/

[16] S. Marchesini et al., "SHARP: A Distributed, GPU-Based Ptychographic Solver, " LBNL-1003977, 2016.

[17] T. Bicer et al., "Rapid Tomographic Image Reconstruction via Large-Scale Parallelization, " Euro-Par, 2015.

[18] A. Sarje et all., "Massively Parallel X-Ray Scatering Simulations," SC, 2012.

[19] PySpark Internals, URL https://cwiki.apache.org/confluence/display/SPARK/PySpark+Internals

[20] Y. Jia t al., "Caffe: Convolutional Architecture for Fast Feature Embedding," ICM, 2014.

[21] M. Abadi et al., "TensorFlow: Large-Scale Machine Learning on Heterogenous Distributed Systems, White Paper, 2015. URL http://tensorflow.org

[22] R. Hegerl and W. Hoppe, "Dynamische Theorie der Kristallstrukturanalyse durch Electronenbeugung im inhomogenen Primarstrahlwellenfeld," Phys. Chem. 7, 1970.

[23] J. M. Rodenburg et al., "Hard-X-Ray Lensless Imaging of Extended Objects, " Physical Review Letters 98, 2007.

[24] P. Thibault et al., "High-Resolution Scanning X-ray Diffraction Microscopy, " Science 321, 2008.

[25] P. Thibault et al., "Probe retrieval in ptychographic coherent diffractive imaging, " Ultramicroscopy 109, 2009.

[26] K. Giewekemeyer, "A study on new approaches in coherent x-ray microscopy of biological speciments, " PhD thesis, Gottingen series in x-ray physics, Volume 5, 2011.

[27] V. Elser, "Phase retrieval by iterated projections, " J. Opt. Soc. Am. A, 2003.

[28] J.R. Fienup, "Phase retrieval algorithms: a comparison, " Appl. Opt. 21, 1982.

[29] H.H.Bauschke, P.L. Combettes, and D.R.Luke, "Hybrid projection-reflection method for phase retrieval, " J. Opt. Soc Am. A 20, 2003.

[30] D. R. Luke, "Relaxed average alternating reflections for difraction imaging. Inverse Problems 21, 2005.

[31] K. Nugent, "Coherent methods in the x-ray sciences," Adv. Phys. 59, 2010.

[32] X. Huang et al., "Fly-scan ptychography," Scientific Reports, X-Ray Microscopy, 2015.

[33] J. Dean et al., "Large Scale Distributed Deep Networks," NIPS, 2012.

[34] P. Moritz et al., "SparkNet: Training Deep Networks in Spark," ICLR, 2016.

[35] CaffeOnSpark, URL https://github.com/yahoo/CaffeOnSpark

[36] TensorSpark, URL https://github.com/adatao/tensorspark

[37] MVAPICH, URL http://mvapich.cse.ohio-state.edu/

[38] G.Fox et al., "Towards an Understanding of Facets and Examplars of Big Data Applications, ", In Proc. of 20 Years of Beowulf, 2014.

[39] G. Fox et al., "HPC-ABDC High Performance Computing Enchnced Apche Big Data Stack, " CCGrid, 2015.

[40] STREAM 2015 and STREAM 2016, URL http://streamingsystems.org

[41] N.Chaimov et al., "Scaling Spark on HPC Systems, " HPDC, 2016.

[42] J. Liu et al., "H5Spark: Bridging the I/O Gap between Spark and Scientific Data Formats on HPC Systems, " CUG, 2016.

[43] S. Sehrish, J.Kowalkowski, and M.Paterno, "Exploring the Performance of Spark for a Scientific Use Case, " FERMILAB-CONF-16-072-CD, 2016.

[44] A. Gittens et al., "A Multi-Platform Evaluation of the Ranomized CX Low-Rank Matrix Factorization in Spark, " IPDPS, 2016.