MPIgnite: An MPI-Like Language for Apache Spark

Extended Abstract

Brandon L. Morris
Auburn University
Auburn, Alabama
blm0026@auburn.edu

Anthony Skjellum*†
Auburn University
Auburn, Alabama
skjellum@auburn.edu

ABSTRACT

Scale-out parallel processing based on MPI is a 25-year-old standard with at least another decade of preceding history of enabling technologies in the High Performance Computing community. Newer frameworks such as MapReduce, Hadoop, and Spark represent industrial scalable computing solutions that have received broad adoption because of their comparative simplicity of use, applicability to relevant problems, and ability to harness scalable, distributed re- sources. We introduce a new framework, called MPIgnite, that serves as an augmentation of the popular cloud data processing engine Apache Spark. Our work introduces message passing into the strictly data-parallel platform to create a blend of a high level environment with the granular capabilities common in MPI applications.

KEYWORDS

MPI, Spark, data-parallel, task-parallel, Scala, peer-to-peer communication, parallel closures

ACM Reference format:
Brandon L. Morris and Anthony Skjellum. 2017. MPIgnite: An MPI-Like Language for Apache Spark. In *Proceedings of EuroMPI/USA Conference, Chicago, Illinois USA, September 2017 (EuroMPI/USA 2017),* 2 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In recent years, the demand for powerful data analysis and processing tools has exploded with the surge of available data and advancement of commodity "off the shelf" computational power. However, despite MPI's dominance in the traditional field of High Performance Computing (HPC), widespread interest in the standard has not followed the "data deluge trend" [3]. Instead, communities of developers and companies have created their own software solutions to large-scale computational problems, with the particular emphasis on a high-level interface that is simpler for developers without domain expertise to utilize, even at the cost of performance. Some examples of such approaches include MapReduce [2], Google's TensorFlow [1], and the open source project Apache Spark [4].

These applications generally excel in domains that heavily exploit data parallelism, and have little if any need for task parallelism that is at the core of MPI. Our main contribution focuses on incorporating some of the fundamental features of message passing into the high-level Apache Spark framework. We do so by leveraging the existing infrastructure within the project whenever possible and without interfering with its basic capabilities. Our work demonstrates that such a marriage of data- and task-parallel paradigms is possible in popular computing frameworks, and doing so creates an interesting programming environment that is more flexible to suit a programmers' preferences and the specific problem domain.

2 APPROACH

Our primary goal in this work is to incorporate peer-to-peer message passing within the Apache Spark framework. Differing from work that attempts to run Spark on HPC infrastructure or expand MPI to emulate a MapReduce model, we aimed at harmoniously amalgamating core MPI functions within Spark. We specifically chose to break backwards compatibility with the standard, both in the interest of providing a fresh perspective to the established model, as well as to facilitate development in the atypical environment. Most notably, MPIgnite transfers actual objects in messages, which is natural in the Scala language. To handle non-blocking receives, we also utilized Scala's standard `Future` class normally used for delayed execution. As a prototype implementation, we also made specific decisions to deviate from standard implementations to facilitate development. Scalability and performance were not a primary concern, but were left for future work.

Another goal of our work was to present a programming model that would be familiar to users of both Spark and MPI, the features of each could be used interchangeably. To do this, we built on Spark's `parallelize` method that transforms data sets into distributable RDDs that are fundamental to Spark. We created a method `parallelizeFunc` that accepts anonymous or named functions that take a single parameter: the MPIgnite communicator, or `SparkComm`. The communicator is heavily influenced by MPI communicators and serves as the central object for communication operations (send, receive, etc.).

Function closures passed into the `parallelizeFunc` method serve as the basis of the parallel program. They can be as long or as short as appropriate, and even be chained together to execute in succession. The end of a single or chain of parallel closures

*Corresponding author
†Present address: SimCenter, University of Tennessee, Chattanooga, TN; email: *tony-skjellum@utc.edu.*

EuroMPI/USA 2017, September 2017, Chicago, Illinois USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

MPIgnite	MPI
comm.send(rec, tag, data)	MPI_Send
comm.receive[T](sender, tag): T	MPI_Recv
comm.receiveAsync[T](sender, tag): Future[T]	MPI_Irecv
Await.result(f: Future[T]): T	MPI_Wait
comm.getRank	MPI_Comm_rank
comm.getSize	MPI_Comm_size
comm.split(color, key): SparkComm	MPI_Comm_split
comm.broadcast[T](root, data[1]): T	MPI_Bcast
comm.allReduce[T](data, f(a, b): T): T	MPI_Allreduce

Figure 1: Comparison of the MPIgnite and MPI function signatures

serves as an implicit barrier in the main application. In our current implementation, these closures do not accept arguments other than the required `SparkComm` communicator.This is not a deficiency since values in the outer scope can be referenced in the parallel section, effectively serving as arguments if necessary. Once a closure is parallelized, it can be executed with the `execute` method, that can take an integer as the number of desired processes. The result of the `execute` method returns an array of values (one for each process) returned by the closure, the type of which can be parameterized by a type argument to the `parallelizeFunc` method.

3 EXAMPLES

Below is an example of matrix-vector multiplication with a 2D decomposition. The code is in Scala and would run inside a typical Spark application, where the `sc` variable refers to the `SparkContext` required for all Spark applications. For brevity, the rank values are used for the values of the matrix and vector elements, though any value in the surrounding scope of the closure can be referenced.

Listing 1: Matrix-vector multiplication with 2D data decomposition

```scala
sc.parallelizeFunc((world: SparkComm) => {
  val worldRank = world.getRank
  val row = world.split(worldRank / 3,
    worldRank)
  val col = world.split(worldRank % 3,
    worldRank)
  val a = worldRank + 1
  val rowRank = row.getRank
  val colRank = col.getRank

  // Distribute the vector to the diagonal
  if (rowRank == row.getSize - 1)
    row.send(col.getRank, 0, 1+col.getRank)
  val x_row = if (rowRank == colRank)
    Some(row.receive[Int](
      row.getSize - 1, 0))
    else None
  val multiplied = x_row match {
    case Some(x) =>
      col.broadcast[Int](colRank, x)
      x * a
    case None =>
      a * col.broadcast[Int](rowRank)
  }
  val result = row.allReduce[Int](
    multiplied, (a: Int, b: Int) => a + b)
}).execute(9)
```

4 CONCLUSIONS

We integrated a core facet of traditional HPC, message passing, into the popular cloud computing framework Apache Spark. We did so in such a way that maintains the original capabilities of the framework, introducing an environment that will expand possibilities for traditional MPI developers to utilize sophisticated data parallel infrastructure and high level language concepts. In addition, cloud developers can use our framework to leverage well-studied algorithms and techniques of HPC.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grants Nos. 1562659 and 1229282. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

We acknowledge the previous work and contributions of Mr. Jared Ramsey in his MS thesis at Auburn that motivated this work. Dr. Jonathan Dursi's blog [3] was a strong motivator for this work.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2015).

[2] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51 (2004), 107–113.

[3] Jonathan Dursi. 2015. HPC is dying, and MPI is killing it. (2015). `http://www.dursi.ca/hpc-is-dying-and-mpi-is-killing-it/`.

[4] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. *HotCloud* 10, 10-10 (2010), 95.