Tools and Frameworks for Big Learning in Scala:

Leveraging the Language for High Productivity and Performance

Heather Miller
EPFL, Switzerland
`heather.miller@epfl.ch`

Philipp Haller
EPFL, Switzerland and
Stanford University
`philipp.haller@epfl.ch`

Martin Odersky
EPFL, Switzerland
`martin.odersky@epfl.ch`

Abstract

Implementing machine learning algorithms for large data, such as the Web graph and social networks, is challenging. Even though much research has focused on making sequential algorithms more scalable, their running times continue to be prohibitively long. Meanwhile, parallelization remains a formidable challenge for this class of problems, despite frameworks like MapReduce which hide much of the associated complexity. We present three ongoing efforts within our team, previously presented at venues in other fields, which aim to make it easier for machine learning researchers and practitioners alike to quickly implement and experiment with their algorithms in a parallel or distributed setting. Furthermore, we hope to highlight some of the language features unique to the Scala programming language in the treatment of our frameworks, in an effort to show how these features can be used to produce efficient and correct parallel systems more easily than ever before.

1 Introduction

There is a growing need to facilitate the analysis of large data. Machine learning (ML) has provided elegant and sophisticated solutions to many complex problems on a small scale, which if ported to large scale problems, could open up new applications and avenues of research for numerous fields. Unfortunately, ML research efforts are routinely limited by the complexity and running time of sequential algorithms. Despite their success in other contexts, parallel programming paradigms like MapReduce/Hadoop are, in many ways, too limited to express the charicteristics of problems which frequently arise in fields like ML. Inspired by these limitations, we present several projects which, in different ways, aim to make it easier for ML researchers and practitioners to implement and experiment with their algorithms in a parallel or distributed setting.

The goals of this paper are two-fold; we aim to showcase some of our tools, libraries, and frameworks suited to large-scale parallel and distributed learning, meanwhile highlighting language features that the Scala programming language integrates in a unique way, and which we argue make using and developing parallel systems much more easily achievable.

We first present three projects suited for large-scale ML. Two of which have been recently presented at non-ML oriented venues: Menthor [1], a framework for distributed graph-processing, and Scala's Parallel Collections [2], a library which provides parallel bulk operations on generic collection types, such as Arrays, Maps, and Sets. And finally, OptiML [3], a parallel domain-specific language (DSL) for machine learning on heterogeneous hardware platforms[1].

[1]As a member of the Scala team at EPFL, Heather has worked on Scala parallel collections, and is also one of the primary authors of the Menthor ML framework. As a member of Stanford's PPL, Philipp works on debugging tools for the Delite DSL framework, as a member of the Scala team at EPFL, he works on the Scala language and (concurrency) libraries.

Finally, we hope to show how each of these projects rely on a unique combination of interesting language features, which in many respects are responsible for making their implementations possible, and we note the existence of other related projects like Spark [4] and FACTORIE [5] who have also chosen Scala for its language features.

2 Our Tools, Libraries, and Frameworks for Big Learning

2.1 Menthor

Menthor [1] is a framework and abstraction for parallelizing (and distributing) iterative algorithms, that is well-suited to operate on graphs. It is implemented as a normal library in Scala, and as such, requires no additional compiler plugin.

Within the Menthor programming model, data is represented as a graph; vertices represent atomic data items, and edges represent relationships between these data items[2]. Data items stored inside of vertices are iteratively updated in *globally synchronized supersteps*, inspired by Valiant's Bulk Synchronous Parallel model [6] and Google's Pregel [7], via an `update` method, until some convergence condition is satisfied.

Additionally, supersteps can be created by a composition of so-called *substeps*. A set of *substep combinators* available within the framework allows composing substeps to form chains of computations within supersteps, which essentially structures an iteration into several phases, where each phase may be either parallel or sequential. The combinators available include `then` steps (and variants such as `thenUntil`) which roughly correspond to a `map` or `foreach` combinator, and `crunch` which corresponds to a `reduce` combinator, commonly found in functional programming. A principal difference, however, between a standard functional combinator and the substep combinators provided by the Menthor framework is that the result of a substep combinator is made available as an incoming message to vertices at the start of the next substep, via the `incoming` field. In particular, the `crunch` combinator ensures that its result is available to all vertices, while all other combinators make their results available to a targeted subset of vertices, making it easy to express message-passing algorithms on graphs. A more deailed treatment of the programming model can be be found in [1] along with examples and a performance evaluation.

2.2 Parallel Collections

Parallel Collections [2] is a normal library included in Scala's standard distribution; as such, it requires no additional compiler plugin, and ships with the standard Scala compiler. It can be thought of as simply a parallel version of the standard collections library with exactly the same interface.

Like Scala's sequential collections library, data structures (collections) implemented in Scala's parallel collections framework, each come with dozens of functional operations, such as `fold`, `reduce`, `filter`, `foreach`, and `map`. Thus, if one can express their algorithm in a functional style, it can be automatically parallelized by simply using the Scala parallel collections framework.

Usage is simple– most collections in Scala's standard library include a parallel counterpart. A sequential collection can be converted to a parallel version simply by calling its `par` method. Subsequently, all bulk operations performed on the parallel version are executed in parallel.

The parallel collections framework was also designed to be generic; users are able to define their own collections, which automatically inherit all of the parallel methods supported within the framework. Extensibility and the design of the framework is further elaborated upon in [2] along with performance evaluations.

2.3 OptiML

OptiML [3] is a domain-specific language (DSL) for machine learning. It is embedded in Scala which means that a Scala compiler is used to compile DSL programs.[3] The OptiML language itself is a restricted subset of Scala. Programs can be compiled for a variety of parallel hardware platforms, including CMPs

[2]For data types that are not typically represented by graphs, like images or video for example, a pixel can be represented as a vertex with edges connecting neighboring pixels.

[3]Currently, OptiML requires the Virtualized Scala compiler which is a small compiler extension supporting language virtualization [8].

(chip multi-processors), GPUs (by automatically generating CUDA code), and eventually even FPGAs and other specialized accelerators. Cluster support is currently not available, but is a topic of ongoing work. Furthermore, compilation employs aggressive domain-specific optimizations, resulting in high-performance generated code which outperforms parallel MATLAB on many common ML kernels.

OptiML makes it easy to express iterative statistical inference problems, such as those conforming to the Statistical Query Model [9] which has been shown to cover a large subset of ML algorithms [10]. Most of these problems are expressed using dense or sparse linear algebra operations which can be parallelized using a large number of fine-grained map-reduce operators. OptiML programs make use of three fundamental data types, Vector, Matrix, and Graph, which support all of the standard linear algebra operations used in most ML algorithms. These data types are polymorphic and are compiled to efficient code leveraging BLAS or GPU support if they are used with scalar values.

OptiML provides high-level control structures tailored to iterative algorithms involving vectors and matrices. Consider the following example using the `untilconverged` operator:

```
untilconverged (mu, tol) { mu =>
  // computing newMu
  newMu
}
```

The `untilconverged` operator takes two parameters `mu` and `tol` as arguments, as well as a function `{ mu => ... newMu }` that takes a value `mu` and returns a value `newMu`. Executing `untilconverged` iterates until the difference between `mu` and `newMu` falls below the tolerance `tol`.

Vectors and matrices offer convenient ways for construction, and can be used with normal arithmetic syntax (e.g., `x(i)` for indexing). In summary, the provided features enable programs resembling pseudocode or scripts. In [3] the authors discuss more extensive examples that illustrate the conciseness of OptiML, while also showing some detailed benchmarks.

3 Scala's Language Features for Big Learning

In the following we present a number of language features that we categorize according to the level of (Scala) expertise required to make effective use of them (basic, intermediate, and advanced). Based on the above categories, we provide an overview of how the language features discussed below, and the tools discussed above in Section 2 come together to affect end users, ML experts, and parallel system architects. For example, the OptiML DSL can be **used** by an ML expert, it can be **implemented** by an intermediate Scala user, but it requires a parallel system architect to **distribute** its run-time system.

- **For comprehensions (basic).** Make it possible to perform implicity parallel bulk operations on parallel collections. For example,
  `for (url <- urls){ url.rank = computePageRank(url) }`
- **Closures (basic).** Used for high-level control structures like `untilconverged` in OptiML and `crunch` in Menthor (see Section 2). Many bulk operations for parallel collections take closures as parameters (e.g., the `find` method takes a closure to filter with as an argument). Moreover, closures enable productive use of immutable data structures, which are essential for providing fault tolerance in Spark [4].
- **By-name parameters (intermediate).** Library methods can take unevaluated blocks of code as arguments, known as by-name parameters. The library or framework can then chose to evaluate the code block at a later point in time. This mechanism is used in Menthor to create chains of computational steps by passing code blocks to substep combinators, like `then` or `crunch`. *By-name parameters enable one to create what seem like custom keywords without extending the language or compiler.*
- **Syntactic shorthands (intermediate).** Scala provides a small number of simple syntactic rewritings which help produce clearer and more readable APIs. For example, the expression `x(i) = x(j) + 1` is rewritten to `x.update(i, x.apply(j) + 1)`. By providing appropriate definitions for the `update` and `apply` methods, OptiML enables standard arithmetic notations for vectors and matrices. *Syntactic shorthands can be used to provide familiar notations for custom data types.*

- **Trait composition (intermediate).** Scala's traits [11] roughly correspond to interfaces in Java (supporting multiple inheritance), except that they can also have concrete methods and fields. Trait composition is used to combine the functionality of multiple traits in a new trait or class. It is used when extending the parallel collections framework with a new, custom collection type, for example, a DNA sequence type. *Traits enable safe and flexible combinations of type hierarchies, resulting in richer ways to extend libraries and frameworks.*

- **Implicits (advanced).** Implicits are used, for example, in Scala's parallel collections to avoid bit rot [12] (complicated and difficult-to-maintain libraries, full of code duplication). In parallel collections, they enable uniform result types for parallel bulk operations: for example, invoking map on a parallel vector returns a parallel vector, as opposed to a less-specific super type. Arguably, this makes parallel collections easier to use by preventing the user from having to use type casts in many cases. *Implicits allow framework implementers to avoid having to write a considerable amount of repetitive boilerplate code.*

- **Delimited continuations (advanced).** Continuations make it possible to suspend computation and to automatically create a closure (see above) to hold the remaining compution within. A popular Scala middleware used for distributing computation, known as Akka, uses continuations for providing futures [13] for lightweight, event-driven actors [14, 15]. *Continuations make it possible to define APIs with what seem like blocking operations, while never actually blocking at all, resulting in extremely lightweight and scalable processing of concurrent tasks.*

- **Specialization (advanced).** In parallel collections, specialization of type parameters can be used to avoid the performance overhead of boxing, the wrapping of primitives on the heap, which leads to an indirection when attempting to access data elements. Specialization avoids this, thus improving cache locality which is especially important on multi-core systems to achieve good scalability. *Specialization is essential for high-performance parallel data structures on the JVM.*

- **Higher-kinded types (advanced).** Makes it possible to avoid the use of concrete type constructors. Used in the DSL framework, Delite [16] to be generic in the type constructors used for code generation. This enables targeting different hardware platforms (e.g., GPUs) without changes to the DSL. *Higher-kinded types allow framework implementers to abstract from concrete type constructors, enabling more code re-use.*

Analogous to the above language features we provide a qualitative comparison according to the level of Scala experience required to use, extend, and architect each framework, respectively, summarized in Table 1.

Using all of the introduced frameworks requires only basic knowledge of Scala, thanks to clear, readable APIs which don't expose users to complex features. The original architecture of Menthor as well as its extension requires intermediate Scala experience, given the use of by-name parameters, syntactic shorthands, and trait composition. The architecture of Scala's parallel collections is based on several advanced features, such as implicits and higher-kinded types. However, only an intermediate level of experience is needed to extend the framework with custom collection types, thanks to its flexible trait hierarchy designed for user extensions.

Implementing the OptiML DSL is feasible with only an intermediate level of Scala experience, thanks to the predefined building blocks of the Delite DSL framework. However, the facilities for code generation and scheduling that Delite provides require an expert in parallelism to build.

4 Conclusion

There is a lot of activity on leveraging Scala's expressiveness and ecosystem for large-scale learning tasks. This paper introduces three projects that are conducted in the context of the Scala team. By focusing on the used language features, we attempt to distill commonalities among the discussed tools and frameworks; at the same time, this allows us to differentiate the tools and frameworks based on their implementation approach. We hope that a categorization of language features according to the level of required Scala experience helps end users, ML experts, and system builders to find features and tools that satisfy their particular needs. We hope that this exposition contributes to a rich exchange among existing and potential users on how to make–and how others have made–best use of Scala for Big Learning.

References

[1] Philipp Haller and Heather Miller. Parallelizing machine learning- functionally: A framework and abstractions for parallel graph processing. In *2nd annual Scala Workshop, Palo Alto, CA, USA, June 3, 2011*, 2011.

[2] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A generic parallel collection framework. In *Proceedings of the 17th International Conference on Parallel Processing, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011*, Lecture Notes in Computer Science, pages 136–147. Springer, 2011.

[3] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616. ACM, June 2011.

[4] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2011-82, Department of Electrical Engineering and Computer Science, University of California at Berkeley, July 2011.

[5] Andrew McCallum, Karl Schultz, and Sameer Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems 22*, pages 1249–1257. 2009.

[6] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[7] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010*, pages 135–146. ACM, June 2010.

[8] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 835–847. ACM, October 2010.

[9] Michael J. Kearns. Efficient noise-tolerant learning from statistical queries. *J. ACM*, 45(6):983–1006, 1998.

[10] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *Proceedings of Neural Information Processing Systems, NIPS 2006*, pages 281–288, 2006.

[11] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 41–57. ACM, October 2005.

[12] Martin Odersky and Adriaan Moors. Fighting bit rot with types (experience report: Scala collections). In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2009, December 15-17, 2009, IIT Kanpur, India*, volume 4, pages 427–451, 2009.

[13] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[14] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Massachusetts, 1986.

[15] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci*, 410(2-3):202–220, 2009.

[16] Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *20th International Conference on Parallel Architectures and Compilation Techniques, PACT 2011*, October 2011.

Framework	**Use**	**Extension**	**Architecture**
Menthor [1]	Basic	Intermediate	Intermediate
Parallel Collections [2]	Basic	Intermediate	Expert
OptiML [3]	Basic	Intermediate	Intermediate
Delite [16]	Intermediate	Expert	Expert

Table 1: Scala experience required to use, extend, and architect frameworks for large-scale learning.