Scalable Bootstrapping for Python

Peter Birsinger
UC Berkeley
peterbir@eecs.berkeley.edu

Richard Xia
UC Berkeley
rxia@eecs.berkeley.edu

Armando Fox
UC Berkeley
fox@cs.berkeley.edu

ABSTRACT

High-level productivity languages such as Python, Matlab, and R are popular choices for scientists doing data analysis. However, for today's increasingly large datasets, applications written in these languages may run too slowly, if at all. In such cases, an experienced programmer must typically rewrite the application in a less-productive performant language such as C or C++, but this work is intricate, tedious, and often non-reusable. To bridge this gap between programmer productivity and performance, we extend an existing framework that uses just-in-time code generation and compilation. This framework uses the SEJITS methodology, (Selective Embedded Just-In-Time Specialization [11]), converting programs written in domain-specific embedded languages (DSELs) to programs in languages suitable for high performance or parallel computation.

We present a Python DSEL for a recently developed, scalable bootstrapping method; the DSEL executes efficiently in a distributed cluster. In previous work [16], Prasad et al. created a DSEL compiler for the same DSEL (with minor differences) to generate OpenMP or Cilk code. In this work, we create a new DSEL compiler which instead emits code to run on Spark [18], a distributed processing framework. Using two example applications of bootstrapping, we show that the resulting distributed code achieves near-perfect strong scaling from 4 to 32 eight-core computers (32 to 256 cores) on datasets up to hundreds of gigabytes in size. With our DSEL, a data scientist can write a single program in serial Python that can run "toy" problems in plain Python, non-toy problems fitting on a single computer in OpenMP or Cilk, and non-toy problems with large datasets on a multi-computer Spark installation.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*; D.3.2 [**Programming Langauges**]: Language Classifications—*concurrent, distributed, and parallel languages*

Keywords

Domain-specific languages; SEJITS; bootstrapping

1. INTRODUCTION

Domain scientists create applications more quickly when using high-level languages such as Python and Matlab, but often must re-write their code in more efficient languages such as C++ and CUDA to obtain the performance they need, or to make use of a dataset too large to fit on a single computer. In other words, they must select between programmer productivity and application performance. Alongside performance, application scalability is a primary concern with today's growing datasets. While some cluster computing frameworks now have a programming interface to a productivity language (e.g. Hadoop with R [3], Spark with Python [17]), this does not relieve programmers of rewriting their application to fit the system's API, or the strengths and weaknesses of the distributed hardware platform.

An existing collection of tools, Selective Embedded Just-In-Time Specialization (SEJITS) [12], bridges this gap between performance and productivity with a methodology for extending productivity languages such as Python with Domain-Specific Embedded Languages (DSELs). A programmer experienced in writing high-performance, parallel code in a lower-level language first designs a DSEL to express a particular problem type or computational pattern. The programmer then uses code-generation and templating techniques to create a just-in-time (JIT) compiler for the DSEL whose output code targets parallel or distributed hardware. The Asp framework [11] (a recursive acronym for "Asp is SEJITS for Python") provides facilities for embedding such DSELs into Python and for creating the compilers themselves in Python; thus, the DSEL is defined as a subset of Python and appears to application programmers as a Python class. Because these DSEL compilers only target a single type of computation, and because they can be expressed in Python themselves, they are much smaller and less complex than standard compilers, typically a few hundred lines of code, despite being able to perform sophisticated optimizations.

Once the DSEL compiler is created, application writers never leave the world of Python. Their code written in the DSEL is JIT-compiled, and the resulting output code is dynamically linked to the application at runtime. To the domain scientist, the experience is that the Python program simply runs faster, and may even benefit from performance portability, if the DSEL compiler can generate code for different hardware targets from the same DSEL.

We present a Python-embedded DSEL for the Bag of Little Bootstraps (BLB) algorithm [13, 14], along with a DSEL compiler to generate distributed BLB applications. Like the standard bootstrap, BLB quantifies the uncertainty of a statistical function's estimate, but the algorithm's structural properties lend themselves to parallelization and distributed computation. Previously, a nearly identical DSEL was created in Python for BLB using the SEJITS methodology, along with a compiler for the DSEL which generates high-performance C++ with either OpenMP or Cilk [16]. Here, we augment the existing Asp framework to create a new DSEL compiler that allows programs written in the same BLB DSEL to run efficiently on the Spark cluster computing system [18]. We explore two diverse real-life applications which testify to the scalabilty of the generated distributed code.

The benefits of our work are threefold. First, a data scientist can write a program in simple sequential Python code that can operate across hundreds of cores and hundreds of gigabytes of data. Second, because our DSEL is almost identical to the one consumed by the existing DSEL compiler that targets a single multicore computer [16], the same application can run efficiently on smaller datasets that fit on a single computer by exploiting multicore parallelism. (Future work would allow the Asp framework to automatically select either the multicore or Spark hardware target depending on hardware availability and dataset size.) Finally, our extensions to Asp that enable generation and execution of Spark code serve as a guide to efficiency programmers for creating DSELs for alternate computational patterns and for targeting DSELs to other distributed computing environments such as Map-Reduce.

2. A DSEL FOR BLB

BLB, similar to the classical bootstrap, quantifies uncertainty on a statistical estimate, but is better suited for a distributed implementation due to its structure. From the input data of size n, it samples without replacement subsamples of size n^γ for typically $0.5 < \gamma \leq 0.9$. This results in a relatively small number of unique points per subsample compared to the total input size n. From each subsample, bootstraps of size n are sampled with replacement, and the statistical estimator function is computed on each bootstrap. The differences between the estimates made on the bootstraps for each subsample are quantified, typically with a standard deviation or variance calculation. The algorithm's output is the average of the error measurements on each of the subsamples.

The following parameters are needed to specify a BLB problem instance:

1. number of subsamples, number of bootstraps, and a subsample size exponent γ, a real number
2. statistical estimator function, a Python function that can use a specific subset of the Python language
3. a reducer function, similarly
4. a specification of the target execution environment: plain Python, Cilk on a multicore computer, OpenMP on a multicore computer, or Spark on a cluster of computers

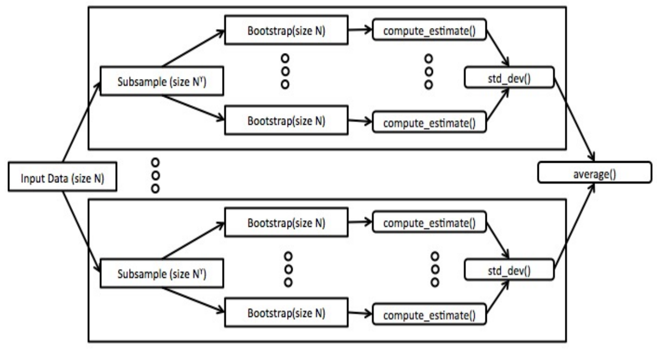

Figure 1: Workflow of BLB. Subsamples are subsampled *without* replacement, while bootstraps are drawn *with* replacement.

5. a set of input data: for Spark, this is specified by a Universal Resource Identifier (URI) for the Hadoop Distributed Filesystem (HDFS) which is used by Spark; for OpenMP or Cilk, a filename.

Our DSEL provides a compact interface in Python to specify these parameters, some of which are set to reasonable defaults if unspecified. The application writer subsets our Python BLB class and defines the estimator and reducer functions using a modest subset of Python, shown in Figure 2. A formal description of the distributed backend's DSEL can be found in the appendix, but generally the DSEL provides enough mathematical operations and control flow for most statistical estimator functions. Due to Scala's static typing, the argument and return types of the estimator and reducer functions must be explicitly specified, something not usually required in Python programs. The application writer does this by setting an instance variable in the BLB class that specifies the type information. Note that the limited subset of Python must be respected only within the estimator and reducer functions; the remainder of the application can use the full Python language.

- Control: `for, while, if`
- Common arithmetic, binary, and boolean operations
- Variable declarations
- Basic array and string operations: `len, range, zip, split`. However, all objects in a Python array must be of the same type, because the Scala code generator treats Python lists and tuples as Scala arrays.
- Conversion among integers, strings and floats

Figure 2: Informal description of the subset of Python available for expressing the estimator and reducer functions.

We also provide a Scala library with several common statistical functions that can be named directly from Python: standard deviation, mean, and dot product. Additionally, since BLB applications often involve machine learning, we include a feature vector class, a compressed feature vector class, and functions to construct these classes from input feature vectors in String format.

The data scientist specifies the input data at runtime by passing in the data file's Hadoop Distributed File System (HDFS) URI. This allows our generated code to operate on files too large to fit on a single node. Supplementary files needed for the computation (e.g. machine learning models) can additionally be included, although these files are read and stored locally, not in a distributed fashion, due to their empirically smaller sizes.

Figure 3 shows an example instance of our BLB class (without type declarations) where the estimator function computes the sum of an array of numbers. This example, aside from the extreme simplicity of the estimator function, is representative of the code that the data scientist would write in order to use our framework.

```
class SumVerifier( BLB ):

    TYPE_DECS =
    (['compute_estimate', [('array', 'double')], 'double'],
     ['reduce_bootstraps',[('array', 'double')],'double'],
     ['average', [('array', 'double')], 'double'])

    def compute_estimate( array ):
        sum = 0.0
        for elem in array:
            sum += elem
        return sum

    def reduce_bootstraps( bootstraps ):
        std_dev(bootstraps)

    def average(subsamples):
        mean(subsamples)
```

Figure 3: Example BLB application to quantify error, in terms of standard deviation, of the sum of an array of numbers.

To summarize, our DSEL allows data scientists to use Python to express a variety of BLB problem instances that can be run as distributed computations across hundreds of cores. We next describe how this DSEL is compiled and the resulting Spark program in Scala is executed.

3. A SEJITS COMPILER FOR THE BLB DSEL

Using the SEJITS approach, we create a JIT compiler for the BLB DSEL by extending the existing Asp infrastructure. The compiler maps a Python BLB instance to a Scala BLB instance suitable to run on at least tens of nodes on the Spark cluster computing system. Spark is similar to Map-Reduce, but performs significantly better on iterative algorithms [18] and uses Scala as the main client-side language.

We contribute an efficient, Scala Spark implementation of the common elements of BLB to encapsulate the data scientist's estimator and reducer functions. Our hand-crafted code reads the input data from HDFS, forms the subsamples and bootstraps, and calls the estimator and reducer functions.

Our distributed implementation of BLB can accomodate a variety of problem types by performing several optimizations transparently to the application writer:

- To allow utilizing large numbers of cores, we parallelize across bootstrap estimate computations, rather than across subsample error estimate computations.
- We replace the standard Java serialization used by Scala with Kryo serialization [6], which is both faster and produces a more compact data representation (up to a factor of 10 smaller [7]).
- We store the intermittent computations in serialized form, which requires more CPU but gives better results on large datasets since more objects can be stored in the cache.
- We automatically select the level of parallelism (number of Spark tasks for a BLB instance) based on the number of nodes in a cluster and the number of cores in each node, opting for double the number of cores in the cluster.
- We broadcast to all nodes at the start of the computation any subsidiary data that is not part of the dataset but which is needed for the computation, such as machine learning models. This can reduce the size of serialized tasks as well as the cost of starting a job [10].

The DSEL compiler combines this "boilerplate" code with Scala code generated from the data scientist's input estimator and reducer functions. Exercising Python's reflection capacities, we parse the estimator and reducer functions' Python Abstract Syntax Trees (ASTs) to convert them to Scala ASTs. The DSEL compiler performs minor optimizations on the Scala ASTs, such as converting for-loops to while-loops, which can greatly improve performance [9, 1]. Finally, the compiler generates optimized Scala code from the optimized Scala ASTs to fill in the hand-coded, BLB Scala skeleton.

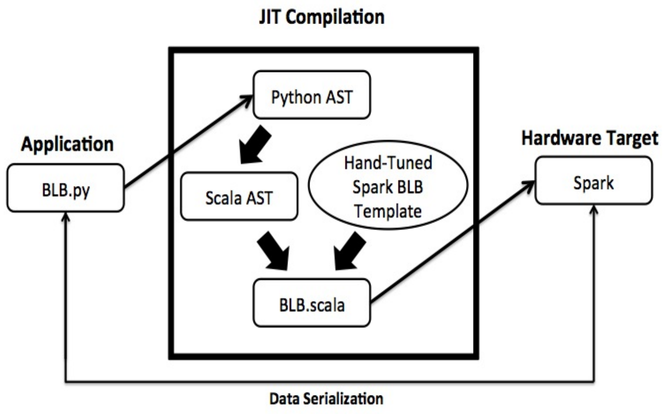

Figure 4: BLB DSEL compiler workflow. The Python BLB application is used to generate Scala code, which when combined with our hand-tuned Spark BLB template, forms a BLB application runnable in a distributed setting on Spark.

The DSEL compiler then packages the complete Scala program inside a Scala object, automatically generating the `main()` method based on the input Python arguments. After runtime compilation with the Scala compiler's optimization flag set, Asp initializes Spark to run on the Scala object by a shell command. Meanwhile, the DSEL compiler stores and tracks compiled DSEL programs so that identical code need not be compiled repeatedly when the application is re-run.

The input arguments for BLB, namely the HDFS URI of the data file, along with the BLB sampling parameters (and possibly the URI of a machine learning model or other supplementary data), are passed from Python to Scala using Apache Avro serialization [2]. After the distributed Scala BLB app runs on the input data from HDFS, the results are returned to Python via Avro once more.

We have supplied Asp with the infrastructure to generate Scala code, initialize Spark, and pass data between Python and Scala. Further details of Asp, in particular how it handles C++, are covered in [12].

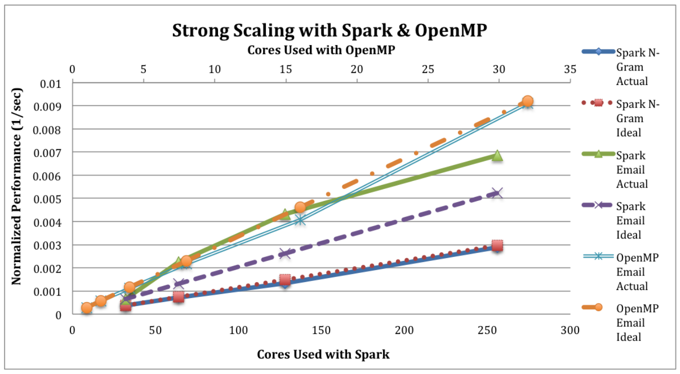

Figure 5: Spark strong scaling results on 32, 64, 128, and 256 cores (4-32 Amazon EC2 m2.4xlarge nodes) on 103,576,600 (34 GB) emails from the Enron email corpus and on 201GB of N-Grams. Also shown are OpenMP scaling results on 126,000 Enron emails obtained from 4 Intel X7560 processors[16].

4. EVALUATION

We have examined two diverse applications of BLB, one relating to machine learning and one involving the Google N-gram dataset. These applications demonstrate not only the ability of our generated code to scale well on large datasets, but also the breadth of statistical computation that can be expressed with our DSEL.

4.1 Email Classifier

We evaluate estimates of a classifier's accuracy on the publicly available Enron email corpus, consisting of approximately 1.2 million emails [15]. We classify emails based on their user defined directory name (e.g. Inbox, Sent, Vacation), and select for each email the most probable class it belongs to. To improve classification efficiency, we take only emails in the top twenty directories, still preserving over 90% of the original dataset. For each email, we create a feature vector using a bag-of-words model; each word denotes a feature and the number of occurrences of that word in the email is the feature's weight. Each feature vector is then of length 139,578, the number of words in the entire corpus. Since most emails contain a very small fraction of the total words, we store each email as a compressed feature vector with two arrays, one for the indices, and one for the non-zero feature weights. We create the machine learning models for our classifier using a Support Vector Machine (SVM) multi-class library [8].

We attempt to estimate the classifier's performance on the entire 1.2 million emails (apart from 10% which we use as training data) by calculating its accuracy rate on a subset of size roughly 20% (215,000 emails) of the original corpus. We run BLB on the 20% subset as well to estimate how widely the exact error rate on the entire corpus varies from our estimate of it based on the classifier's performance on the subset. The actual classification error rate on the entire corpus is 67.74%, while it is only 67.70% on the subset. With the settings of 25 subsamples, 50 bootstraps, and a subsample exponent γ of .7, BLB consistently predicts that the error rate on the entire corpus varies with about a .10% standard deviation from the estimate on the subsample. Considering that the two estimates were only .04% apart, this estimate appears reasonable. Through empirical evidence, we found that the aforementioned sampling parameters were ample to obtain an accurate answer.

The estimator function, as seen in Figure 7, consists of roughly 20 lines of Python, containing a doubly nested for loop and a dot product. The reducer functions, standard deviation and mean, simply call the provided Scala library.

To test the scaling properties of the generated code, we duplicate the entire test data set one hundred fold to create 34 GB of email data. We rent Amazon EC2's high memory quadruple extra large instances, each with 8 cores, to form clusters of 4, 8, 16, and 32 slave nodes. We run BLB with the same sampling parameters of 25 subsamples, 50 bootstraps, and a subsample exponent γ of .7.

Between 4 and 8 nodes, we witness in Figure 5 super-linear scaling as larger numbers of nodes allow more of the dataset and intermittent computations to be stored in memory, as opposed to disk. Near perfect scaling occurs across 8 and 16 nodes. Between 16 and 32 nodes though, we experience moderately sub-linear scaling as a result of dwindling parallelism across 256 cores, but still the completion time for 32 nodes is 10.47 times less than the 4 node result.

For data sets able to fit on a single node, particularly one with many cores, however, the generated OpenMP and Cilk code run more efficiently than the Spark code. Although lacking formal verification, this presumably results from the increased inter-node communication costs of distributed computation. Prasad et. al previously evaluated the generated OpenMP on 126,000 (three orders of magnitude fewer) emails with a nearly identical estimator function and achieved near perfect speedup on a 32-core CPU as seen in Figures 5 and 6. [16]. These results demonstrate the large scalability of our generated code on varying input data sizes.

4.2 N-Gram Word Frequency

We next utilize the Google n-gram [5] dataset to evaluate the calculation of the top ratios of 2-gram occurrence frequencies between decades in the English language. This statistic highlights the decades during which new phrases came into popularity the most rapidly. We calculate for each 2-gram the ratio of its frequency in the current decade and in the previous decade and then take the top 10,000 of

Data Type	Dataset Size	# Subsamples	# Bootstraps	γ	Cores used	Speedup	Platform	Lines of Python
emails	34 GB	25	50	0.7	32-256	1.31x	Spark	24
emails	41MB	25	40	0.7	1-32	0.988x	OpenMP	26
n-grams	201 GB	25	100	0.5	32-256	0.816x	Spark	85

Figure 6: Experiments using BLB for email classifier evaluation and n-gram frequency estimation. For the email corpus, we estimate the dataset size based on the number of emails; we report the size of the feature vectors in compressed format. The OpenMP result was reported in [16] and is presented here for comparison only.

```
class SVMEmailVerifierBLB( BLB ):

    TYPE_DECS =
    (['compute_estimate',
      [('array', 'CompressedFeatureVec'),
       ('array', ('array', 'double'))],
      'double'],
     ['reduce_bootstraps',[('array', 'double')],'double'],
     ['average', [('array', 'double')], 'double'])

    def compute_estimate( feature_vecs, models ):
        errors = 0.0
        num_feature_vecs = 0
        for feature_vec in feature_vecs:
            weight = feature_vec.weight
            num_feature_vecs += weight
            tag = feature_vec.tag
            choice = 0
            max_match = -1.0
            for model in models:
                total = dot( model, feature_vec )
                if total > max_match:
                    choice = index() + 1
                    max_match = total
            if choice != tag:
                errors += weight
        return errors / num_feature_vecs

    def reduce_bootstraps( bootstraps ):
        std_dev(bootstraps)

    def average(subsamples):
        mean(subsamples)
```

Figure 7: BLB instance to evaluate estimates of an email classifier's performance. The Python application is nearly identical for the OpenMP and Spark DSELs.

these ratios for each decade (ignoring all 2-grams below a frequency of .0001%). To improve bootstrapping accuracy, we modify the original dataset so that each n-gram contains the number of occurrences for every year, as opposed to having each n-gram only contain the number of appearances for one year. After filtering out all n-grams containing punctuation or numbers, we are left with 92,290,642 n-grams totaling 38 GB.

The estimator function written in our DSEL performs three nontrivial tasks. After computing the amount of 2-grams in each decade, it uses priority queues, for which equivalent Scala operations are generated, to find the top 10,000 2-gram ratios for each decade. It then averages these 10,000 ratios for each decade, and outputs the array. Notably, an array, as opposed to just a double, may be returned from the estimator, reducer, or average function. The 75 lines of Python used for this statistical computation testify to the large variety of applications our DSEL can support.

We evaluate BLB on a random 10% of our 2-gram dataset, using BLB parameters of 30 subsamples, 40 bootstraps, and a γ of .85. We normalize every average decade ratio with the 1900 ratio, dividing all other ratios by it. We notice from the results that for all decades but 1910 and 1920, the actual ratio was within one predicted standard deviation. The standard deviations increase with later decades because there are more 2-grams in later decades to choose from for subsamples and bootstraps. Additionally, several 2-grams, primarily technology oriented ones such as "NET Framework" and "Windows Server", have enormous ratios and their inclusion can largely skew a sample's average ratio. Interestingly, the highest ratios are 1990 and 2000, presumably due to technology's enormous growth, and additionally 1940 and 1950 presumably due to WWII.

Decade	Actual Ratio	Ratio on 10% Subset	BLB Std. Deviation
1900	1	1	0
1910	1.112	1.585	0.213
1920	0.869	1.413	0.405
1930	1.239	1.252	0.469
1940	1.605	1.841	1.082
1950	1.713	1.341	0.699
1960	0.958	1.434	0.750
1970	1.098	1.459	0.606
1980	1.190	1.418	1.886
1990	1.730	2.008	3.373
2000	3.180	3.657	3.173

Figure 8: The actual average ratio of the top 10,000 2-grams' occurrence frequency between each decade and the previous one, in addition to these ratios on a 10% subset and the BLB predicted standard deviation.

To test the scaling properties of the generated Spark code, we duplicate the dataset repeatedly to obtain 201GB of 2-gram data. We run BLB with the sampling parameters of 25 subsamples, 100 bootstraps, and a subsample exponent γ of .5. We select γ to be .5 to lessen computation time; increasing it would not worsen scaling, as easily partionable additional computational work would only be added. As Figure 5 attests, we obtain close to perfect strong scaling between 4 and 32 nodes (or 32 and 256 cores). We ran these computations again on Amazon EC2's high memory quadruple extra large instances, each with 8 cores. Unfortunately, we were not able to run this application with the

C++ DSEL compiler as it does not support several features in the estimator function.

5. FUTURE WORK

Much further work can be done on our Spark BLB DSEL compiler, in addition to testing it on more applications. Increasing the overlap between its DSEL and that of the C++ compiler's DSEL is a prime goal. Furthermore, we hope that later versions of our compiler can automatically determine the optimal target backend (Spark, OpenMP, or Cilk) based on the BLB instance's properties, most particularly the size of the input data.

The BLB and Spark parameters could similarly be autotuned. The number of subsamples, bootstraps, and the subsample exponent length could be automatically optimized by the SEJITS framework to pinpoint the ideal intersection of efficiency and accuracy for BLB.

Our current compiler only customizes a few of the Spark default settings, and all statically except for the level of parallelism. This leaves room for much innovation. More generally, the type and number of EC2 instances recruited could as well be dynamically computed based on computational workload and a pricing budget.

Lastly, the Python and Scala ASTs could be optimized based on runtime information regarding the input data or the cluster being run on. Additionally, application level optimizations based on assumptions made from the constraints on the DSEL could be made. Both of these kinds of optimizations are displayed by already exisiting SEJITS DSEL compilers [11, 4].

6. CONCLUSION

We have presented a DSEL and a matching compiler that, in the context of the BLB, brings scalable data analysis to Python. This work describes the construction of a DSEL that executes efficiently both in a distributed setting and on a single node. The two real-life applications we present attest to not only the scalability of the generated code, but also to the breadth of applications that the DSEL can support. The generated Spark code achieves near-perfect scaling on hundreds of GBs of data with up to 256 distributed cores across 32 nodes.

We created this DSEL compiler by broadening the already existing Asp framework to enable the execution of Python DSELs on the Spark cluster computing system. The shared BLB DSEL between this DSEL compiler and a previously-built DSEL compiler targeting OpenMP and Cilk facilitates the generation of highly performant code for data ranging from MBs to hundreds of GBs. This work demonstrates the strength of the SEJITS approach, packaging the expertise of performance programmers for wide reuse by productivity programmers, and suggests a means to create additional DSELs for alternate computational patterns and hardware platforms.

7. ACKNOWLEDGMENTS

Shoaib Kamil provided both valuable encouragement and assistance writing the paper. Michael Driscoll, Ameet Talwalkar, and Yahel Ben David contributed helpful feedback on the paper. Additionally, Ameet and Ariel Kleiner assisted with the usage of the BLB algorithm. David Howard and Aakash Prasad developed the OpenMP and Cilk BLB DSEL compiler along with its original interface. Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates Nokia, NVIDIA, Oracle, and Samsung. Research also funded by DARPA Award Number HR0011-12-2-0016. The content of this paper does not necessarily reflect the position or the policy of the US government and no official endorsement should be inferred.

8. REFERENCES

[1] 3 tips for writing performant scala. `http://www.sumologic.com/blog/technology/3-tips-for-writing-performant-scala`.

[2] Apache avro data serialization. `http://avro.apache.org/`.

[3] Apache hadoop. `http://hadoop.apache.org/`.

[4] Asp specializers. `https://github.com/shoaibkamil/asp/wiki/Specializers`.

[5] Google n-gram dataset. `http://storage.googleapis.com/books/ngrams/books/datasetsv2.html`.

[6] Kryo. `https://code.google.com/p/kryo/`.

[7] Kryo benchmarks. `https://code.google.com/p/kryo/wiki/V1Benchmarks`.

[8] Multi-class support vector machine. `http://svmlight.joachims.org/svm_multiclass.html`.

[9] Scala optimize simple for loops compiler issue. `https://issues.scala-lang.org/browse/SI-1338`.

[10] Spark tuning guide. `http://spark-project.org/docs/latest/tuning.html`.

[11] S. Kamil, D. Coetzee, and A. Fox. Bringing parallel performance to python with domain-specific selective embedded just-in-time specialization. In *Python for Scientific Computing Conference (SciPy)*, 2011.

[12] S. A. Kamil. *Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2013.

[13] A. Kleiner, A. Talwalkar, P. Sarkar, and M. Jordan. The big data bootstrap. *arXiv preprint arXiv:1206.6415*, 2012.

[14] A. Kleiner, A. Talwalkar, P. Sarkar, and M. I. Jordan. A scalable bootstrap for massive data. *arXiv preprint arXiv:1112.5016*, 2011.

[15] B. Klimt and Y. Yang. The enron corpus: A new dataset for email classification research. In *Machine Learning: ECML 2004*, pages 217–226. Springer, 2004.

[16] A. Prasad, D. Howard, S. Kamil, and A. Fox. Parallel high performance bootstrapping in python. In *Proc. Eleventh Annual Scientific Computing with Python*, 2012.

[17] M. Zaharia. Spark overview. `http://spark-project.org/docs/latest/index.html`.

[18] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.