A Unified MapReduce Domain-Specific Language for Distributed and Shared Memory Architectures

Daniel Adornes, Dalvan Griebler, Cleverson Ledur, Luiz Gustavo Fernandes
Pontifical Catholic University of Rio Grande do Sul (PUCRS),
Faculty of Informatics (FACIN), Computer Science Graduate Program (PPGCC),
Parallel Application Modeling Group (GMAP).
Av. Ipiranga, 6681 - Building 32 - Porto Alegre - Brazil
{daniel.adornes,dalvan.griebler,cleverson.ledur}@acad.pucrs.br, luiz.fernandes@pucrs.br

***Abstract*—MapReduce is a suitable and efficient parallel programming pattern for processing big data analysis. In recent years, many frameworks/languages have implemented this pattern to achieve high performance in data mining applications, particularly for distributed memory architectures (e.g., clusters). Nevertheless, the industry of processors is now able to offer powerful processing on single machines (e.g., multi-core). Thus, these applications may address the parallelism in another architectural level. The target problems of this paper are code reuse and programming effort reduction since current solutions do not provide a single interface to deal with these two architectural levels. Therefore, we propose a unified domain-specific language in conjunction with transformation rules for code generation for Hadoop and Phoenix++. We selected these frameworks as state-of-the-art MapReduce implementations for distributed and shared memory architectures, respectively. Our solution achieves a programming effort reduction from 41.84% and up to 95.43% without significant performance losses (below the threshold of 3%) compared to Hadoop and Phoenix++.**

Keywords: MapReduce, Domain-Specific Language, Parallel Programming, Effort Evaluation, Performance Evaluation.

I. Introduction

An exponential volume of data is generated by a variety of fields worldwide, for example, social networks, governments, health care, stock market, among others. The so-called *Big Data* is addressed by data analysis applications, which may imply high computational costs. Consequently, high-performance computing is needed to process all data in time. Google initially proposed a solution for improving the performance of these application's domain, by combining *Map* and *Reduce* operations as a single parallel pattern named MapReduce [5]. Since then, the MapReduce has originated many implementations by both industry and academic research. Some of them have achieved great importance, such as Hadoop[1], which is suited for programming in large clusters architectures, and Phoenix++ [13] for programming in multi-core architectures.

MapReduce is a high-level pattern concept for expressing parallelism and taking advantage of different parallel architectures [5]. However, current state-of-the-art implementations impose additional complexities beyond this pattern, requiring developers to deal with low-level programming aspects, such as memory management and network communication. Moreover, there are host language prescriptions imposed by the programming interface of library-based approaches. These aspects motivate a particular language syntax for MapReduce implementation.

This paper proposes a unified domain-specific language to reduce the programming effort and improve code reuse between distributed and shared memory architectures. Code transformation rules are also proposed together with a transformation process aimed at being fully compliant with the key features of original MapReduce solutions.

The contributions are the following:

- A unified MapReduce domain-specific language for parallel and distributed architectures.
- A programming interface approach that significantly reduces the development effort.
- An efficient set of code transformation rules for Hadoop and Phoenix++ without significant performance loss.

The paper is organized as follows. Section II discusses the most important related work. Section III details the proposed domain-specific language. Section IV describes the methodology approached for the evaluation. Section V performs the experiments and evaluates the performance and programming effort results. Finally, Section VI presents the conclusions and future works.

II. Related Work

We aim at providing a unified programming interface to reduce programming effort and allow code reuse between distributed and shared memory architectures. A Domain-Specific Language (DSL) approach allows programmers to focus on specific domains [6]. In this paper, we propose an external DSL consisting of an entirely new language. Related work, in turn, are embedded DSLs, which restricts their programming interface's flexibility and abstraction.

Hadoop was the first widely used MapReduce implementation, aimed at processing large data sets in distributed systems. It provides a Java API for defining *Map* and *Reduce* logic, which are run by distributed computation components over distributed storage components. The distributed storage

[1]http://hadoop.apache.org

DOI reference number: 10.18293/SEKE2015-204

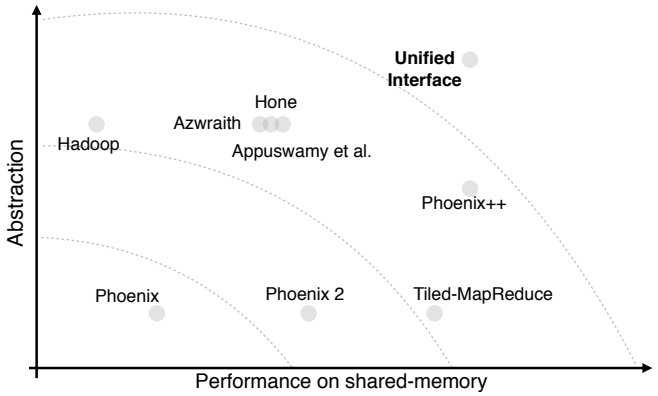

Fig. 1: The relationship graph between abstraction and performance on the programming interface design goals.

components rely on Hadoop Distributed File System (HDFS), which provides the vision of a single file system for large data sets stored on all nodes in the cluster.

Ranger et al. [11] proposed the first version of Phoenix as an optimized implementation of MapReduce for shared-memory architectures, with a C-based programming interface. Yoo et al. [16] evolved the Phoenix project with new optimizations for multi-core architectures with non-uniform memory access. Chen et al. [12], [4] proposed a new implementation of Phoenix, based on a tiled (iterative) approach, named Tiled-MapReduce. Finally, Talbot et al. [13] proposed Phoenix++, consisting of a completely rewritten version of Phoenix, implemented in C++ and taking advantage of the language's capabilities for modularity and code reuse in order to allow a more adaptive programming interface.

Phoenix++ was mainly motivated by the recurrent need of customizations reported by many of its users while working with different applications. Talbot et al. realized that object-oriented capabilities of C++ could be exploited for creating specialized *containers* for storage of intermediate data and *stateful combiners* for storing the cumulative value of associative *reduction* operations (e.g., sum, product).

Concerning to high-performance, Phoenix++ outperforms all its predecessors, previously mentioned. Also, compared to Hadoop, Phoenix++ achieves a 28.5x speedup while executing on a single machine (not distributed). It motivated us to use Hadoop for distributed memory and Phoenix++ for shared-memory multi-core architectures.

Recently, some researches [15], [2], [10] worked on improving Hadoop's performance on the single-node level, mainly by avoiding some internal mechanisms not needed for non-distributed environments (e.g., message passing and replication) and harnessing the computational power of multi-core. These researches also kept the Hadoop's programming interface, thus not adding a new abstraction layer. Their evaluations, though, show that Phoenix++ still outperforms in the single-node level.

Some important implementations addressing heterogeneous architectures, with CPU and GPU, were also analyzed but are still not covered by our transformation process. From these, Grex [3] is the state-of-the-art, providing a specific API for controlling the MapReduce phases and the way data structures are stored in the different memory levels of GPUs.

Through a relationship graph, Figure 1 demonstrates our understanding of the related work achievements concerning abstraction and performance. It is possible to visualize that most programming interfaces designed for distributed architectures provide higher abstraction compared to those for multi-core. Such difference is related to programming aspects and their goals. In shared memory, programmers are required to deal with low-level mechanisms such as pointers and memory allocation in order to maximize the usage of very restricted resources. By other hand, resources are much less restricted in distributed architectures, allowing Hadoop's interface to be favored by using Java as host language.

Moreover, Figure 1 also justifies our choices for generating MapReduce code. Hadoop and Phoenix++ are the best alternatives for our design principles in terms of abstraction and performance. The following section discusses these and other design aspects for the proposed solution.

III. The Proposed Domain-Specific Language

A unified MapReduce programming interface is proposed in conjunction with code transformation rules for Phoenix++ and Hadoop MapReduce.

The proposed interface is inspired on the building block syntax proposed by Griebler et al. [8], [7], since it demonstrated significant effort reduction for the Master/Slave parallel pattern. Our interface however is not built over a third-part language as C or C++, being based on an own language instead. The different programming languages (Java and C++) and the very specific syntaxes for Phoenix++ and Hadoop code led us to decide for an own language in order to maximize abstraction. A C++ programming interface provided by the Hadoop project, called Hadoop Pipes[2], was initially considered but later discarded due to absence of documentation and for looking as a discontinued project.

The interface's structure consists of an outer *@MapReduce* block and two inner *@Map* and *@Reduce* blocks, as detailed on listing 1 and grammar 1.

```
1 @MapReduce<NAME, K_IN, V_IN, K_OUT, V_OUT, K_DIST>{
2   @Map(key, value){
3     // Map code logic
4   }
5   @Reduce(key, values){
6     // Reduce code logic
7   }
8 }
```

Listing 1: Interface's structure.

The *@MapReduce* block always requires six parameters, namely *NAME*, *K_IN*, *V_IN*, *K_OUT*, *V_OUT* and *K_DIST*.

The *NAME* parameter is any user-defined name, which is used for identifying the MapReduce process and further transforming the code for Java and C++ classes.

[2]http://wiki.apache.org/hadoop/C++WordCount

⟨*Map*⟩ ::= '`@Map`' '`(`' ⟨*key*⟩ , ⟨*value*⟩ '`)`' '`{`' { ⟨*cmd*⟩* ⟨*EmitCall*⟩ ⟨*cmd*⟩* } '`}`'

⟨*Reduce*⟩ ::= '`@Reduce`' '`(`' ⟨*key*⟩ , ⟨*values*⟩ '`)`' '`{`' { ⟨*cmd*⟩* ⟨*EmitCall*⟩ ⟨*cmd*⟩* } '`}`' | '`@SumReducer`' | '`@IdentityReducer`'

⟨*MapReduce*⟩ ::= '`@MapReduce`' '`<`' ⟨*mapreduce-params*⟩ '`>`' '`{`' ⟨*Map*⟩ ⟨*Reduce*⟩ '`}`'

Grammar 1: Structure's grammar

The *K_IN*, *V_IN*, *K_OUT* and *V_OUT* parameters are used to define the *<key/value>* input and output types, respectively. In other words, these parameters define which type of raw data is initially read by the MapReduce process and which type of reduced data is produced by it at the end.

The *K_DIST* parameter, in turn, is used for defining the keys distribution, whether **:**, **:k* or *1:1*. It is used by Phoenix++ [13] for employing memory-optimized data structure for intermediate *<key/value>* pairs, taking advantage of applications in which the number of keys to be emitted is known in advance.

The inner blocks, *@Map* and *@Reduce*, must be programmed by the user in order to define the core logic of the given MapReduce application. The *@Map* block receives a *<key/value>* input pair from which to compute the *<key/value>* intermediate pairs. Finally, the *@Reduce* block receives all mapped values for each key, this is a *<key/values>* pair, and computes the final reduced *<key/value>* pair by key. Both blocks are provided with an *emit* function (grammar 2), which for *@Map* block represents the function to emit intermediate *<key/value>* pairs and for *@Reduce* block represents the function to emit the final reduced value for a given key.

⟨*EmitCall*⟩ ::= '`emit`' '`(`' ⟨*key*⟩ , ⟨*value*⟩ '`)`'

Grammar 2: The emit function's grammar

One additional characteristic of the *@Reduce* block is that it can be replaced by a single *@SumReducer* directive with no block code (grammar 1), which indicates that a simple sum operation must be performed over all values of each key. Another option is the *@IdentityReducer* directive, which indicates that no reduction needs to be performed. Both default options are also provided by Hadoop and Phoenix++, since these are common reduce logics for MapReduce applications. Nevertheless, whenever the provided default reducers do not fit the need, a customized reducer can be implemented, as demonstrated in listing 2 for a sample multiplicand reducer.

```
@Reduce(key, values){
  double product = 1
  for(int i=0; i < length(values); i++)
    product *= values[i]
  emit(key, product)
}
```

Listing 2: Multiplicand reducer with proposed interface.

Finally, listing 3 demonstrates the code of a Histogram application with the proposed interface. This sample implementation uses the *@Type* directive to define a variable type *pixel* that stores the RGB values for each pixel in the processed image. Then, the *@MapReduce* directive defines the name Histogram and the four variable types for *<key, value>* input and output pairs. The **:768* indicates that it is known in advance that a maximum of 768 distinct keys will be emitted by the *Map* phase. This information optimizes the data structure used by the generated Phoenix++ code to hold the intermediate *<key, value>* pairs. Finally, a *@Map* block defines that *map* phase will emit value 1 for each occurrence of a given color in the RGB of the pixel being processed, and the *@SumReducer* defines that reduction will be performed as a simple sum operation over all emitted values for each distinct key.

```
@Type pixel(r: ushort, g: ushort, b: ushort)
@MapReduce<Histogram, long, pixel, int, ulonglong,
        "*:768">{
  @Map(key, p){
    emit(p.b, 1)
    emit(p.g+256, 1)
    emit(p.r+512, 1)
  }
  @SumReducer
}
```

Listing 3: Histogram with proposed interface.

A. Interface components and transformation rules

For developing the *@Map* and *@Reduce* logics the programmer is provided with a set of proposed interface's components, which comprehends variable types, built-in functions and flow control structures. Each of these components has an associated transformation rule, through which its equivalent component in Hadoop and Phoenix++ can be later generated.

Variable types can also be custom types defined by the programmer with the *@Type* keyword, which are translated to C++ *structs* for Phoenix++ and Java classes implementing the *WritableComparable* interface for Hadoop. The resulting Java classes, particularly, include *getters* and *setters* methods, besides some other methods whose implementation is required by *WritableComparable* interface.

Moreover, whenever a custom type is defined for input data (*V_IN*) in Hadoop, a complete implementation of a subclass of *FileInputFormat* and another subclass of *RecordReader* is required. It is particularly needed in order to instruct Hadoop on how to split and distribute the input data among *Map* tasks. Nonetheless, it causes applications developed with Hadoop to reach a considerable amount of code. The generated subclass of *RecordReader* considers each line of an input file to represent a single instance of the given custom type.

Also, whenever a custom type is used for output values (*V_OUT*), Phoenix++ requires the implementation of a custom *associative_combiner*, which in turn is most likely to perform a simple sum for internal attributes of the custom type. By assuming this, the unified interface still allows *@SumReducer* directive even if output values are of a custom type. In this case, the code transformation is defined for the correspondent *associative_combiner* and *Reducer* class of Phoenix++ and Hadoop respectively.

Finally, Phoenix++ requires specific types (*structs*) and a complex *split* logic for text processing applications. In the proposed interface, whenever the type *Text* is chosen as input value (*V_IN*), the transformation rules automatically include such components in the C++ generated code.

B. Transformation Process

Transformation rules are applied through a transformation process, whose stages are described as follows:

- **First stage** - The process starts by generating *imports* (for Hadoop Java code) and *includes* (for Phoenix++ C++ code) always required by any application. It consists of base libraries of these frameworks.
- **Second stage** - The process then continues by transforming the *@MapReduce* block and its *@Map* and *@Reduce* inner blocks. At this same stage, custom types *@Type* may have been provided by the programmer being then also transformed. Ultimately, global variables, external to the *@MapReduce* block, may have also been defined by the programmer and are also transformed in this second stage.
- **Third stage** - This stage addresses transformations derived from the input and output keys and values, interpreted in the second stage (e.g., text processing components previously mentioned).
- **Fourth stage** - This stage transforms the variable types defined in the blocks' signature and also internally to these blocks.
- **Fifth stage** - Ultimately, the fifth stage transforms the functions defined externally to the MapReduce blocks or internally to custom types.

The proposed process is based on Aho et al. [1], thus consisting of language recognition, analysis and code generation. Language recognition phase comprehends the interpretation through lexical, syntactic and semantic analysis. Lexical analysis validates the compliance with the proposed components then producing tokens. The syntactic analysis uses the identified tokens to check the grammar language and report syntax errors. The semantic analysis in turn checks how components are disposed throughout the whole code. An overview of the transformation flow is shown in figure 2. Effective transformations and code generation are proposed as future work (section VI).

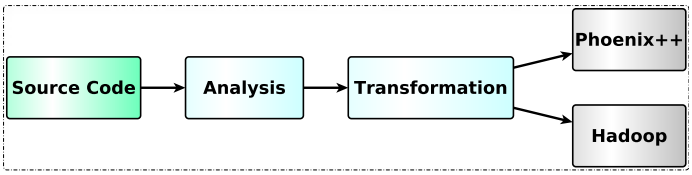

Fig. 2: Domain-Specific Language Flow.

Along the language recognition phase, an AST (Abstract Syntax Tree) is created. An AST is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construction present in the source code. AST creation is bottom-up because the nodes addition starts from the smaller tokens and patterns. Finally, AST stores the identified tokens for later use in the code generation phase, which then traverses the tokens in the AST in order to generate new code in the target language.

IV. Methodology

We evaluated our DSL using two approaches. First we evaluated its interface using SLOCCount[3] to measure programming effort. The second approach consisted of performance results. Five different applications with specific peculiarities, namely Word Count, Word Length, Histogram, K-means and Linear Regression, were implemented with the proposed interface and generated through the transformation rules for Phoenix++ and Hadoop. This applications were also implemented purely in Phoenix++ and Hadoop.

The main peculiarity we looked for while choosing the sample applications was the key distribution. Word Count demonstrates the **:** distribution, whereas Word Length, Histogram, K-means and Linear Regression demonstrate the **:k* distribution. Additionally, other peculiarities are also covered by the selected sample applications, such as custom types and custom combiners.

The Matrix Multiplication and PCA (Principal Component Analysis applications) would fit the *1:1* distribution, however would also require more programming controls for Phoenix++ generated code beyond the abstraction aimed by the proposed interface and with no equivalent functionality in Hadoop.

A. Effort Evaluation

For programming effort measurement it was used the SLOCCount suite, also used by Griebler et al. [7] for the evaluation of DSL-POPP and by a set of other researches (e.g., [9], [14]). SLOCCount[3] is a software measurement tool, which counts the physical source lines of code (SLOC), ignoring empty lines and comments. It also estimates development time, cost and effort based on the original Basic COCOMO[4] model.

The suite supports a wide range of both old and modern programming languages (e.g., C++ and Java), which are naturally inferred by SLOCCount and thus used for measurement. For our unified interface, we selected C++ because it has similar syntax.

B. Performance Evaluation

For evaluating performance, the workload for Word Count and Word Length was a 2Gb text file, for Histogram, a 1.41 Gb image with 468,750,000 pixels and for Linear Regression the workload was a 500Mb file. For Kmeans, no input file is required, since number of points, means, clusters and dimensions are parametrized through command line or assumed to the default values of 100,000, 100, 3 and 1,000, respectively, which were considered for our tests. All workloads are available at the Phoenix++ project's on-line repository[5].

[3]http://www.dwheeler.com/sloccount/sloccount.html
[4]http://www.dwheeler.com/sloccount/sloccount.html#cocomo
[5]https://github.com/kozyraki/phoenix

For performance evaluation of generated Phoenix++, we used a multi-core system with a 2.3 GHz Intel Core i7 processor, four cores with Hyper-Threading and 16Gb of DRAM, whereas for performance evaluation of generated Hadoop, we used 8 nodes of a cluster, where each node is equipped with a 2.4 GHz Intel Xeon Six-Core E5645 processor and 24Gb of DRAM. The cluster sums 192 cores, where the nodes are interconnected by 2 Gigabit-Ethernet networks and 2 InfiniBand networks.

In order to obtain the arithmetic means, 30 execution times were collected for each sample application such as described in Tables II and I, and Figures 4 and 3. For running Hadoop applications, we used a synthetic script [6] and copied to HDFS all data so that all cluster nodes had access.

V. Results

As described in section IV, we evaluated the proposed DSL using two approaches for measuring programming effort and performance. Tables I and II show the mean execution time for each sample application for the code transformed from our proposed unified interface and for the code developed directly from Phoenix++ and Hadoop, respectively. Figures 3 and 4 graphically demonstrate these same measurements.

TABLE I: Original and generated Phoenix++.

	WC	WL	Histogram	Kmeans	LR
Original	5.38	4.02	2.83	5.98	0.62
Generated	5.37	3.99	2.87	6.09	0.63
Difference	-0.27%	-0.9%	1.4%	1.7%	0.3%

TABLE II: Original and generated Hadoop.

	WC	WL	Histogram	Kmeans	LR
Original	36.24	26.36	21.87	51.36	5.97
Generated	37.22	26.48	22.42	50.59	6.01
Difference	2.63%	0.45%	2.45%	-1.52%	0.76%

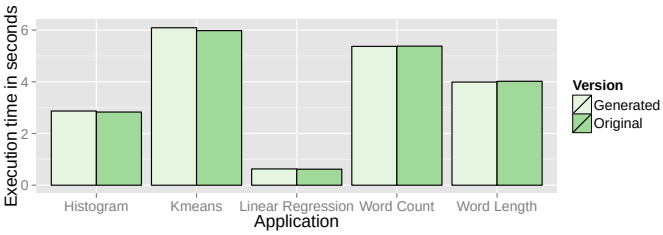

Fig. 3: Original and generated Phoenix++.

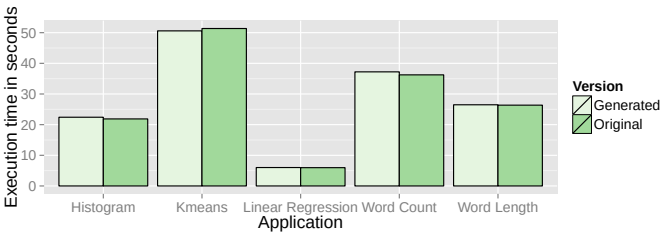

Fig. 4: Original and generated Hadoop.

[6]https://github.com/mvneves/hadoop-deploy

Through tables I and II and figures 3 and 4 it is possible to visualize the negligible difference (less than 3%) between the execution time of the generated and original versions for the two frameworks. Performance losses are considerably avoided as a direct result of the effective coverage of performance components by the transformation rules.

Tables III and IV and figures 5 and 6 show the SLOC and cost measurements. It is possible to observe that the difference between the measurements of SLOC and Cost is negligible, which confirms the approach used by COCOMO model.

A significant SLOC reduction can be observed for *Word Count* and *Word Length* applications compared to Phoenix++ code, which take advantage of the specific components for text processing applications. For such applications, Phoenix++ requires a wide set of mechanisms whose need is then identified in advance by the proposed transformation rules, being it transparent while developing with the proposed unified interface.

TABLE III: SLOC count reduction

Application	**Phoenix++**	**Hadoop**	**Unified Interface**	**Reduction compared to Phoenix++**	**Reduction compared to Hadoop**
WordCount	89	27	8	91.01%	70.37%
WordLength	95	33	14	85.26%	57.58%
Histogram	22	170	9	59.09%	94.71%
K-means	98	244	57	41.84%	76.64%
Linear Regression	31	171	18	41.94%	89.47%
	67	**129**	**21.2**	**63.83%**	**77.75%**

TABLE IV: Cost estimate reduction

Application	**Phoenix++**	**Hadoop**	**Unified Interface**	**Reduction compared to Phoenix++**	**Reduction compared to Hadoop**
WordCount	$2,131.00	$609.00	$170.00	92.02%	72.09%
WordLength	$2,282.00	$752.00	$306.00	86.59%	59.31%
Histogram	$491	$4,204.00	$192.00	60.90%	95.43%
K-means	$2,357.00	$6,143.00	$1,334.00	43.40%	78.28%
Linear Regression	$704.00	$4,229.00	$398.00	43.47%	90.59%
	$ 1,593.00	**$ 3,187.20**	**$ 480.00**	**65.28%**	**79.14%**

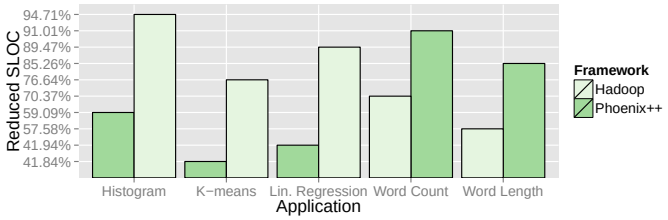

Fig. 5: SLOC count reduction

Compared to Hadoop, the *Histogram*, *K-means* and *Linear Regression* applications achieved greater SLOC reduction mainly for the amount of code required to treat custom types (subclasses of *WritableComparable*) in Hadoop.

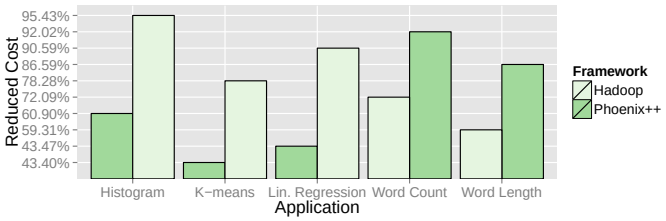

Fig. 6: Cost estimate reduction

K-means also takes advantage over Phoenix++ by completely avoiding code for custom combiner, however many functions are required by the sample implementation, which causes little SLOC reduction with the unified interface.

VI. Conclusions

From selecting Phoenix++ and Hadoop as the state-of-the-art solutions for shared-memory and distributed architectures, respectively, this work proposes a solution for abstracting MapReduce programming without losing the performance optimizations of these selected implementations. Such objective is achieved through a unified MapReduce programming interface, proposed in conjunction with a comprehensive set of transformation rules for Phoenix++ and Hadoop.

Except for a specific data locality configuration for NUMA systems provided by Phoenix++, the transformation rules are effective in covering from custom types to custom functions, custom combiners, default reducers, different key distributions and text processing components, covering thus all components needed from the selected sample applications. Moreover, performance losses are successfully avoided (difference of less than 3%) and SLOC and cost reduction indicates that programmers' productivity can be considerably increased.

Some advantages and main contributions are the reuse of code between different architectures and the possibility of expanding the coverage of the transformation rules to other MapReduce solutions and architectural levels.

A limitation is that programmers are still required to implement the code to call the MapReduce process, thus being required to know C++ and/or Java. However, some on-line services, such as Amazon's Elastic MapReduce[7] (EMR), require only the Hadoop MapReduce implementation, abstracting the invocation code from developers. Nonetheless, we conclude that the SLOC and cost reduction achieved by the proposed interface compensate such limitation.

As future work we plan to expand the transformation rules in order to cover MapReduce solutions such as Grex [3] for heterogeneous parallel architectures. Finally, we also visualize an expansion of the DSL's programming interface, particularly by adding more built-in functions and variable types.

VII. Acknowledgments

This work was supported by FAPERGS (Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul), CAPES (Coordenação de Aperfeiçoamento Pessoal de Nível Superior), FACIN (Faculdade de Informática) and PPGCC (Programa de Pós-Graduação em Ciência da Computação).

[7] http://aws.amazon.com/elasticmapreduce

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs Scale-out for Hadoop: Time to Rethink? In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 20:1–20:13, Santa Clara, CA, October 2013. ACM.

[3] C. Basaran and K.-D. Kang. Grex: An efficient MapReduce Framework for Graphics Processing Units. *J. Parallel Distrib. Comput.*, 73(4):522–533, May 2013.

[4] R. Chen and H. Chen. Tiled-MapReduce: Efficient and Flexible MapReduce Processing on Multicore with Tiling. *ACM Trans. Archit. Code Optim.*, 10(1):3:1–3:30, April 2013.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, Berkeley, CA, USA, December 2004. USENIX Association.

[6] M. Fowler. *Domain-Specific Languages*. Addison-Wesley, Boston, USA, 2010.

[7] D. Griebler, D. Adornes, and L. G. Fernandes. Performance and Usability Evaluation of a Pattern-Oriented Parallel Programming Interface for Multi-Core Architectures. In *The 26th International Conference on Software Engineering & Knowledge Engineering*, pages 25–30, Vancouver, Canada, July 2014. Knowledge Systems Institute Graduate School.

[8] D. Griebler and L. G. Fernandes. Towards a Domain-Specific Language for Patterns-Oriented Parallel Programming. In *Programming Languages - 17th Brazilian Symposium - SBLP*, volume 8129 of *Lecture Notes in Computer Science*, pages 105–119, Brasilia, Brazil, October 2013. Springer Berlin Heidelberg.

[9] M. Hertz, Y. Feng, and E. D. Berger. Garbage Collection Without Paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 143–153, New York, NY, USA, 2005. ACM.

[10] K. A. Kumar, J. Gluck, A. Deshpande, and J. Lin. Optimization Techniques for "Scaling Down" Hadoop on Multi-Core, Shared-Memory Systems. In *Proceedings of the 17th International Conference on Extending Database Technology*, EDBT '14, pages 13–24, Athens, Greece, 2014. Open Proceedings.

[11] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[12] H. C. Rong Chen and B. Zang. Tiled-MapReduce: Optimizing Resource Usages of Data-Parallel Applications on Multicore with Tiling. In *Proc. of the 19th Int'l Conference on Parallel Architectures and Compilation Techniques*, page 523–534, Vienna, Austria, September 2010.

[13] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: Modular MapReduce for Shared-Memory Systems. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 9–16, San Jose, California, USA, May 2011. ACM.

[14] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282, New York, NY, USA, August 2014. ACM.

[15] Z. Xiao, H. Chen, and B. Zang. A Hierarchical Approach to Maximizing MapReduce Efficiency. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 167–168, Washington, DC, USA, October 2011. IEEE Computer Society.

[16] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 198–207, Washington, DC, USA, October 2009. IEEE Computer Society.