A First View of Exedra: A Domain-Specific Language for Large Graph Analytics Workflows

Miyuru Dayarathna
Department of Computer Science, Tokyo Institute of Technology,
2-12-1 Ookayama, Meguro-ku,
Tokyo 152-8552, Japan
dayarathna.m.aa@m.titech.ac.jp

Toyotaro Suzumura
Department of Computer Science, Tokyo Institute of Technology/
IBM Research - Tokyo
2-12-1 Ookayama, Meguro-ku,
Tokyo 152-8552, Japan
suzumura@cs.titech.ac.jp

ABSTRACT

In recent years, many programming models, software libraries, and middleware have appeared for processing large graphs of various forms. However, there exists a significant usability gap between the graph analysis scientists, and High Performance Computing (HPC) application programmers due to the complexity of HPC graph analysis software. In this paper we provide a basic view of Exedra, a domain-specific language (DSL) for large graph analysis in which we aim to eliminate the aforementioned complexities. Exedra consists of high level language constructs for specifying different graph analysis tasks on distributed environments. We implemented Exedra DSL on a scalable graph analysis platform called Dipper. Dipper uses Igraph/R interface for creating graph analysis workflows which in turn gets translated to Exedra statements. Exedra statements are interpreted by Dipper interpreter, and gets mapped to user specified libraries/middleware. Exedra DSL allows for synthesize of graph algorithms that are more efficient compared to bare use of graph libraries while maintaining a standard interface that could use even future graph analysis software. We evaluated Exedra's feasibility for expressing graph analysis tasks by running Dipper on a cluster of four nodes. We observed that Dipper has the ability of reducing the time taken for graph analysis when the workflow was distributed on all four nodes despite the communication, and data format conversion overhead of the Dipper framework.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data mining*
; D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*

General Terms

Languages, Design, Algorithms, Performance, Standardization

WWW 2013 Companion, May 13–17, 2013, Rio de Janeiro, Brazil.
ACM 978-1-4503-2038-2/13/05.

Keywords

Large graph data analysis, domain-specific language, exascale, program synthesis, programming techniques, workflow

1. INTRODUCTION

Recently, areas such as social network analysis, web mining, cyber security, etc. have found applications that require processing large graphs with billions of vertices, and billions of edges. These applications require enormous computational power, and resources which makes them harder to run on single computer. For example, finding the PageRank [26] of a large web graph, finding communities, path traversal, etc. have higher orders of time complexities. Relational databases are less effective in implementing such graph analysis [36].

Special classes of software have emerged to process these data on distributed computing infrastructures such as clusters, and super computers. Pegasus [18], Pregel [24], GBase [17], Trinity [32], Hama [31], Giraph [13], HipG [20], PBGL [14], ScaleGraph [10], KDT [23], etc. are few examples for the large body of research on design and implementation of graph analysis middleware and software platforms targeted for large graph analysis in high performance computing systems. These software provide means for modeling large graphs and allow conducting analysis on them.

Exascale systems [11] which are expected to appear in the time frame 2018-2020 will revolutionize data analysis technologies. However, programming on such Exascale systems will have to operate on a diverse range of hardware such as multi-core processors, hardware accelerators, etc. Furthermore, massive levels of parallelism, increase in system faults, and complex, multi-component software environment are major challenges the Exascale applications will face. There are different path ways for implementing data processing systems in Exascale environments. First such method is use of a standard general purpose programming paradigm and adapt it to different architectures. E.g., MPI is widely used programming model for programming parallel systems. Therefore, we can either emulate MPI and different devices or combine MPI with device specific languages to create graph data processing applications. In the same way we can use novel general purpose parallel programming languages such as PGAS languages (UPC [12], X10 [4], Chapel

[3], etc.) to program data analysis tasks on Exascale systems.

However, a major drawback associated with the aforementioned approaches is that each one of them have their own solution. User must be veteran of the general purpose language rather than the application he/she is interested of. We need unified means of doing large graph processing in future Exascale systems that will enable broad scientific community to leverage the benefits of Exascale. Our approach for programming large graph analysis on Exascale environments (which is currently less studied) is Domain-specific languages [25]. Domain-specific languages (DSLs) are programming languages tailored to a specific application domain, and they represent majority of the programming languages. They exploit domain knowledge for productivity and efficiency. Use of DSLs offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages used for graph analysis.

We introduce Exedra: a domain-specific language for conducting large graph analysis on future exascale systems. Unlike current DSLs for graph analysis our focus is to create a complete DSL for specifying graph analysis workflows. Through Exedra we define a complete language for specifying graph representation, and analytics. We apply this language to a graph processing platform that we developed called Dipper. Dipper allows for specifying large graph analysis tasks using R [28] which is a programming language for statistical computing. Dipper extends the famous Igraph [6] graph analysis package for specifying the large graph analysis tasks. Users can run graph analysis tasks using their familiar Igraph/R [33][6] interface which gets translated to Exedra statements by Dipper. These statements are interpreted by the Exedra interpreter and they get mapped in to the underlying high performance libraries.

Our contributions in this paper can be listed as follows,

- *Exedra* : We introduce the design of a domain-specific language for programming Exascale graph analysis tasks.
- *Dipper* : We implement Exedra DSL on an experimental graph analysis platform by extending the famous Igraph/R interface.
- *Performance Evaluation* : We report a preliminary performance evaluation of our approach by conducting graph analysis tasks using Dipper.

The rest of the paper is organized as follows. In the next section we provide related work for Exedra and Dipper. Next, we describe the language syntax and the design decisions that we took while devising the Exedra in Section 3. We describe the design and implementation of Dipper in Section 4. Then in Section 5 we make a preliminary evaluation of the performance of Dipper platform. We discuss the results we obtained by implementing Dipper in Section 6. We conclude the paper in Section 7.

2. RELATED WORK

Exedra provides a unique programming interface for large graph analysis with the emphasize on programmer productivity. It descends from large graph analysis software on High Performance Computing (HPC) environments yet significantly deviates from the current approaches. Several other research have been conducted on creating DSLs for graph analysis, and increasing graph analysis productivity. Yet we believe we are the first to introduce a DSL for creating unified large graph analysis workflows to increase graph analyst's productivity targeting Exascale systems.

Green-Marl is a DSL designed specifically for graph analysis algorithms [16]. Compared to Exedra which is an interpreted language, Green-Marl is a compiled language. Green-Marl code gets compiled to C++ code while Exedra statements are directed to the underlying library/middleware. Furthermore, graph data structures in Green-Marl are immutable. Yet, in Exedra graph's content can be changed during execution. Currently Green-Marl does not cover broad set of analytics such as clustering, community detection, etc. which are available in Exedra.

We adapt similar syntax to Igraph/R [6][33] package when creating Exedra DSL. Igraph is written in C and we use Igraph/R to denote its R interface [33]. The main reason for this design decision was due to the fact that Igraph/R package is currently widely used by the community of complex network analysts and it would be easier for domain experts to utilize the same syntax when they work with Exedra. However, there are limitations for the Igraph/R grammar such as specification of graph traversal operations that we address in Exedra.

Gremlin [30][15][29] and SPARQL [34] are two examples for DSLs that can be used for specifying graph analysis tasks. Gremlin is a graph traversal language and is completely oriented toward specifying queries over graph data. Yet, Gremlin does not have constructs to express specific algorithms such as degree distribution calculation, graph clustering, pattern matching, etc. Exedra improves over Gremlin in this aspect which we believe is very important for allowing users to specify a variety of graph analysis algorithms. SPARQL on the other hand is the norm for accessing RDF graphs. Yet it also lacks the support for expressing graph analysis algorithms. SPARQL's syntax is closer to SQL while Exedra adopts R syntax used in Igraph/R package.

Knowledge Discovery Toolbox (KDT) is graph analysis library written in Python which aims for high level exploratory graph analytics by domain experts and lower level specification of graph operations by the developers of graph analytics [23]. KDT is built on top of the CombinatorialBLAS [1] which is a C++ library for combinatorial computational kernels. Current version of KDT is interfacing only CombinatorialBLAS, and it has not been extended for other back ends. Hence KDT can only utilize the computational functions offered by Combinatorial BLAS.

Neptune is a DSL based on Ruby programming language which automates configuration and deployment of scientific software frameworks over disparate cloud computing systems [2]. Neptune integrates support for MPI, MapReduce, Unified Parallel C (UPC) [5], X10 [4], etc. Neptune is closely related to Exedra, however Exedra's focus in only on large graphs.

Das *et al.* created Ricardo [7] a data analysis platform which allows analysts to work on huge data sets using R language. Yet, Ricardo does only integration of R with Hadoop.

Dipper integrates multiple programming libraries/middleware for large graph analytics.

ScaleGraph is an X10 library for billion scale graph analytics [10][9]. The library is completely written in X10 which is a programming language developed by IBM Research with the goal of simplifying the programming model in a way that increases the programming productivity for future Extreme scale systems. Compared to ScaleGraph, Exedra is a higher level graph analytics platform which aims for unifying graph analysis activities.

Kennedy *et al.* described a strategy called Telescoping languages [19] where they developed a framework for automatic generation of optimizing compilers for new domain-specific languages. They generate a library aware optimizer which uses the information gathered during preprocessing to carry out fast an effective optimization of high level scripts. Different from them our focus is only on one single domain-specific language. However, we do a library aware optimization a similarity that we share with their approach.

A summary of graph analysis software described in this section is shown in Table 1.

Table 1: A summary of graph analysis software.

	Supports Distributed Processing?	Mutable Graph Data Structure?	Interpreted?	Graph analysis library?
Igraph	No	Yes	No	Yes
KDT	Yes	Yes	Yes	Yes
ScaleGraph	Yes	Yes	No	Yes
SPARQL	No	Yes	Yes	No
Gremlin	No	Yes	Yes	No
Green-Marl	No	No	No	No
Exedra	Yes	Yes	Yes	No

3. EXEDRA DOMAIN-SPECIFIC LANGUAGE

As mentioned earlier, Exedra is a domain-specific language for large graph analysis. We designed the language following data flow paradigm. Exedra is an interpreted language similar to several related domain-specific languages such as R, Gremlin, etc. We also made an important design decision that to follow the Igraph/R syntax as mentioned previously. Exedra has language constructs for defining computation compartments. Computations are conducted on different compartments that makes a data flow path. A compartment is an independent block of computation. It can either receive data from another compartment or instantiate data from itself. After processing the data the compartment hands over the data to another compartment in the flow graph.

Figure 1 shows some examples for codes written in Exedra depicting the language's core language constructs. An Exedra program consists of one or more compartments. Every Exedra compartment starts with `exedra:begin`, and ends with `exedra:end`, thus the simplest program is of the form Figure 1(a). User can pass parameters to a compartment.

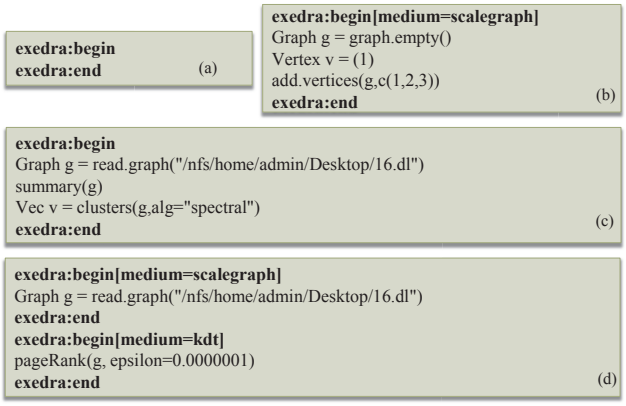

Figure 1: Example Exedra code segments.

One of the most important parameters is *medium*. The medium is the library/middleware that the Exedra compartment runs on. Currently we implemented support for ScaleGraph, KDT, and R mediums. While ScaleGraph, and KDT are large graph processing libraries, we provide R language also as a medium in Exedra since non-graph computations can be specified in R. Furthermore, if a compartment has been marked with execution medium as R, its contents are not interpreted by Exedra interpreter, rather directed to R interpreter directly.

If any parameter has not been provided (as in the case with compartments in Figure 1 (a) and (c)) the default medium's language configured with Exedra interpreter is used as the language of interpreted output.

Furthermore, variables in Exedra have an associated type. For example, the variable g in Figure 1 (c) is of type `Graph`. The language has support for Graph, Arr (Array), Vec (Vector), String, Int (Integer), Float, Double, Long types. Note that type system is only valid on non-R Exedra compartments. Comments on Exedra begins with `#` and Exedra supports only single line comments (see Figure 5).

If an Exedra program specifies more than one compartment (E.g., Figure 1 (d)) they are executed in sequential order. In Figure 1 (d), first the graph loading happens in ScaleGraph compartment. Once the ScaleGraph compartment finishes its execution, PageRank is calculated on KDT compartment. Such type of multiple compartments can be used to leverage several libraries/middleware to conduct a computation that cannot be conducted single handed on any of those software. For example in the scenario shown in Figure 1 (d), ScaleGraph is capable of loading EdgeList files while present version of KDT does not support such feature.

In this paper we describe two scenario workflows for graph analysis and how they can be represented in Exedra. The first scenario is privacy preservation of graphs and social networks [35]. In a social network an adversary can know any subgraph around a certain individual A [21]. If such a subgraph has been identified in the anonymized graph with high probability, user A has high identity disclosure risk. Liu *et al.* described a technique for minimally modifying the graph to protect the identity of each individual involved. In brief their technique involves two steps. First they construct an

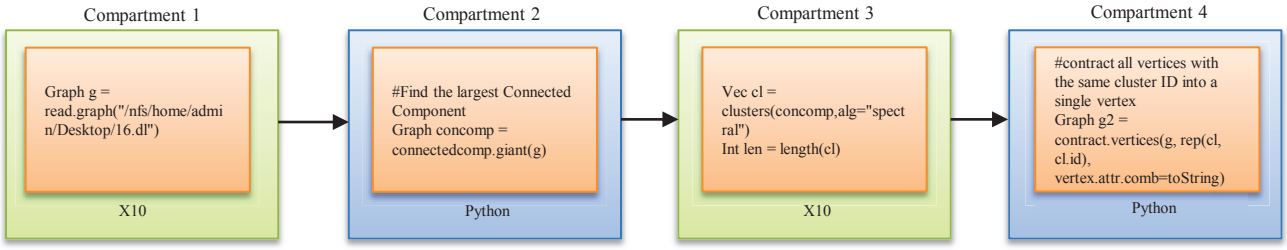

Figure 3: Scenario 2.

```
exedra:begin[medium=scalegraph]
Graph g = read.graph("/nfs/home/admin/Desktop/16.dl")
Vec d = degree(g,mode=c("total"))
exedra:end

exedra:begin[medium=R]
len = length(d) - 1
from = c()
to = c()

repeat{ flg = 0
                flg2 = 0
                for(i in 1:len){
                        if(d[i] < 0){
                                flg = 1
                                break
                        }else if(d[i] != 0){
                                flg2 = 1
                        }
                }
                if((flg == 1)||(flg2 == 0)){
                        break
                }
                randval = sample(1:len, 1)
                d[randval] = 0

                largest = 0
                counter = 0
                len = length(d)
                for(i in 1:len){
                        if(randval != i){
                                if(d[i] > largest){
                                        largest = i
                                }
                        }
                }

                d[largest] = d[largest] - 1
                from = append(from, c(randval))
                to = append(to, c(largest))
}
relation = data.frame(from,to)

exedra:end

exedra:begin[medium=scalegraph]
Graph result = graph.data.frame(relation, directed=TRUE)
exedra:end
```

Figure 2: Example Exedra code for scenario 1.

anonymous degree sequence `d'` for a given degree sequence `d` of a graph `G`. Next, they use `d'` to construct an anonymized graph `G'`. More details on their technique is available from [21]. We have coded the second step of this process as shown in Figure 2.

This involves three Exedra compartments. The first compartment loads the graph G and calculates its degree sequence d. Next, the R compartment creates the anonymized graph using the degree sequence d (Note that for simplicity we choose the same degree sequence `d` as the `d'` anonymized sequence). Next, the data frame named `relation` constructed by the R compartment is fed to the third compartment which constructs a ScaleGraph Graph instance. This is an example for an Exedra code that involves multiple compartments. Also, the second compartment does not involve processing any graph instances.

The second example scenario we have chosen is a Miniworkflow that was introduced by Lugowski *et al.* while introducing the KDT library [23]. The workflow is illustrated in Figure 4 which conducts spectral clustering on the giant component of a graph. How this workflow gets executed on Dipper is shown in Figure 3. The workflow first loads the graph of edge list format from disk. This is done in a ScaleGraph compartment because KDT does not support this feature currently. Next, the graph's largest connected component is calculated on a KDT compartment (Currently the largest connected component calculation is not supported in ScaleGraph). The largest connected component is sent to a ScaleGraph compartment to conduct spectral clustering which is not implemented in KDT yet. Finally, all the vertices of each cluster are contracted in a KDT compartment. The Exedra code for this workflow is shown in Figure 5.

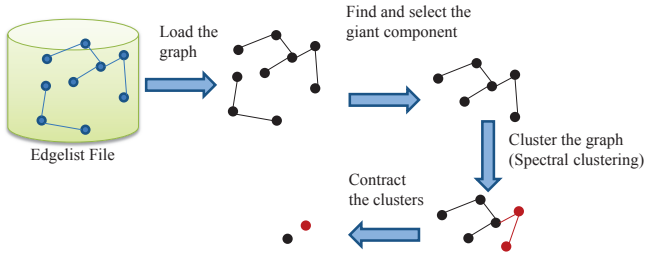

Figure 4: Scenario 2.

```
exedra:begin[medium=scalegraph]
Graph g = read.graph("/nfs/home/admin/Desktop/16.dl")
exedra:end

exedra:begin[medium=kdt]
#Find the largest Connected Component
Graph concomp = connectedcomp.giant(g)
exedra:end

exedra:begin[medium=scalegraph]
Vec cl = clusters(concomp,alg="spectral")
exedra:end

exedra:begin[medium=kdt]
#contract all vertices with the same cluster ID into a single vertex
Graph g2 = contract.vertices(g, rep(cl, cl.id), vertex.attr.comb=toString)
exedra:end
```

Figure 5: Example Exedra code for scenario 2.

4. THE DESIGN, AND IMPLEMENTATION OF DIPPER

We have implemented a large graph processing platform called Dipper which uses Exedra as the language for programming graph analysis tasks. This infrastructure is written in C/C++. System architecture of Dipper is shown in Figure 6. It should be noted that even though Dipper accepts instructions given in Exedra, the end-user of the system writes the graph analysis computations in Igraph/R syntax.

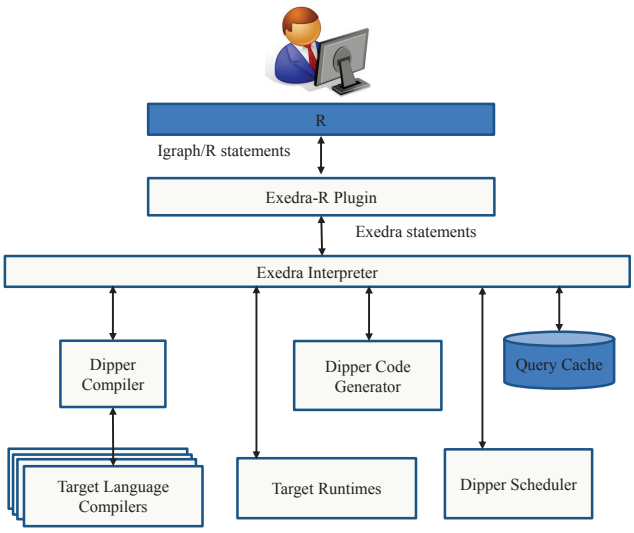

Figure 6: System Architecture of Dipper.

We have developed an R extension (Exedra-R Plugin) by extending the Igraph/R package. Currently this extension converts a subset of the Igraph/R instructions in to Exedra instructions (The remaining Igraph/R instructions are interpreted by the Igraph/R extension itself). The generated Exedra statements are fed to the Exedra Interpreter component. It consists of an Exedra parser written using popular ANTLR [27] parser generator. The Exedra parser identifies each available compartment on the Exedra program. It also checks in its Query Cache whether same query (i.e., Exedra program) has been executed recently. If the same query had been executed recently, it will just reuse the program binaries available on Query Cache rather than moving to Dipper's code generation and compilation phases. If the query was not available in query cache the Exedra Interpreter proceeds to the code generation phase. In this phase, once the target medium is identified each statement in the program are translated to target medium's language (E.g., X10, Python, etc.) or operating instructions. Next, these instructions are added to predefined compartments by the Dipper Code Generator.

A compartment is a standalone software module that runs as an application of its target language. For example an X10 compartment is an X10 application. Compartment has server/client interface; and each compartment communicates with its parent (i.e., predecessor in a compartment chain), and its child (i.e., successor) using a custom protocol written over TCP/IP. The location of a compartment is determined by the Dipper Code Generator. The location where a compartment gets executed is determined during the code generation time.

Once the Dipper Code Generator's code generation phase completes we, get a complete graph analysis workflow that is made out of different graph analysis software components. Next, the Dipper Compiler compiles this workflow using each compartment's target programming language. This step is needed only for compiled languages such as X10, C/C++, Java, etc. Once the complete workflow is ready to be executed; the Dipper Scheduler instantiates the compartments, and launches the workflow.

5. EVALUATION

We evaluated the performance of running a Dipper workflow, and also the performance of running a similar workflow in a single library. We chose the second scenario described in the Section 3 as the example workflow. Specifically we report the execution time for execution of the workflow shown on Figure 5, and also its KDT implementation. The KDT implementation's code is shown in Figure 7.

```
import sys
import kdt

bigG = kdt.DiGraph.load("/tmp/data/qh768.mtx")
bigG.spOnes()

#Finding the largest component
comp = bigG.connComp()
giantCompRoot = comp.hist().argmax()
G = bigG.subgraph(mask=(comp==giantCompRoot))

#Clustering
clus, markovG = G.cluster('Markov', addSelfLoops=True, expansion=2,
inflation=2, prunelimit=0.00001)

#Contracting
smallG = G.contract(clusterParents=clus)
```

Figure 7: KDT version of the example scenario 2.

We use a compute cluster of 4 nodes each with Intel®Core™i7-2600K CPU @ 3.40GHz, 24 cores, 15GB RAM, 8KB L2 cache, 200GB hard drive to run both the Dipper workflow and the KDT workflow. Each node was installed with Linux

CentOS kernel 2.6.18, X10 version 2.2.2, and Python 2.7.1. We used KDT version 0.2 during the evaluation. All the four nodes were connected through 1 Gigabit Ethernet. We used the University of Florida's sparse matrices during this evaluation [8]. The details of the graphs that were used are listed on Table 2. While we could have used larger graphs during this initial evaluation, we used the graphs mentioned in the Table 2 because the KDT implementation that we used has been evaluated using graphs of such scale.

Table 2: Experiment graph data sets

graph	vertices	edges
bfwb398	398	2910
dw256A	512	2,480
qh768	768	2,934
qh882	882	3354
qh1484	1484	6100

The first half of the experiments were run on a single node. Our objective was to get an idea of how the performance varies for each experiment setup with different graph characteristics. The performance results are shown in Figures 8 and 9. Next, in the second step we distributed the Dipper compartments on to 4 different nodes (of the same specification described above). Each node had one Dipper compartment. However, this step could not be performed on KDT since the application was made to run on single node. The results of the second step are shown in Figure 10. All the results shown in Figures 8, 9, and 10 are three times average values.

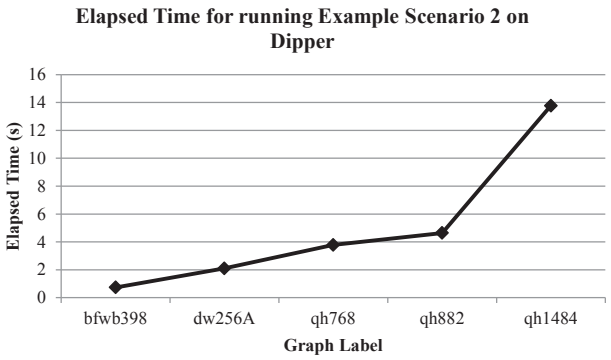

Figure 8: Experiment results of running Dipper.

6. RESULTS AND DISCUSSION

From the results in Figures 8 and 9 we saw that increase of the graph size increases the elapsed time of the analysis process. However, Dipper spent more time in completing the task because it had the overhead of format conversion. Furthermore, the workflow used Spectral clustering rather than

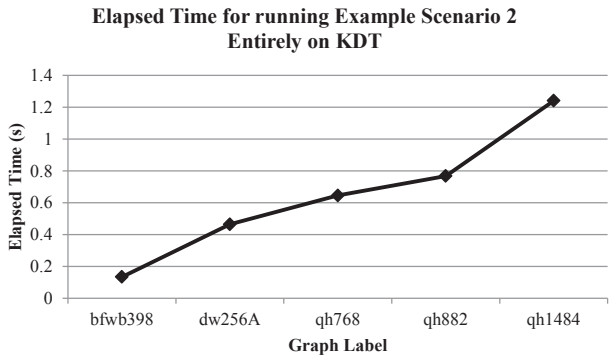

Figure 9: Experiment results of running KDT workflow.

the Markov clustering used in KDT. The distributed version of the experiment indicated that Dipper has the ability of reducing the time consumed for completing the process when distributed (Even though this involves additional network overhead during communication between compartments on different nodes). Note that we were not able to get results for `dw256A` graph in distributed environment due to an error thrown in KDT compartment.

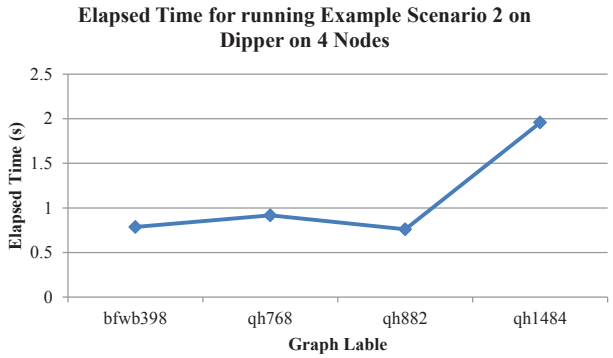

Figure 10: Experiment results of running Dipper distributed on 4 nodes.

There are several limitations for current Exedra Language, and the Dipper framework that we list down here. Current version of Exedra grammar is limited to graph reading, and graph analysis algorithms such as degree distribution calculation, clustering, largest component, etc. Current version of Exedra grammar does not support graph traversal operations. Furthermore, we have implemented compartments for ScaleGraph, and KDT libraries only which restricts us using other mediums. Despite the benefit that Exedra provides for constructing large graph analysis workflows, and distributing graph computations, communication between different compartments introduces an additional overhead of data format conversion. We are investigating on different approaches to minimize the overhead involved with format conversion. Furthermore, the current Dipper framework is not able to withstand the changes made to the libraries/middleware on which the compartments run on.

7. CONCLUSION

Mining valuable information from large graphs is essential in the era of Exascale which is predicted to appear in next five or seven years time period. However, most of the analysis, and mining of such large graph data organized as workflows need coordinated effort of several software components since it is hard to find a single library/middleware that provides all the features expected by users. This is because the rapid evolution of the field of large graph analysis, and the plethora of software written for this purpose have different applications. In this introductory paper we described Exedra, a domain-specific language that we believe will provide a unique way of specifying large graph analysis workflows. The workflows get translated to the most appropriate middleware/library that it should get executed. This provides ability of constructing scalable graph analysis workflows that is not possible using single library or middleware.

Current Exedra grammar has limited support for a variety of graph analysis tasks. We hope to implement such analysis processes in future versions of Exedra grammar. Furthermore, we hope to support different other types of mediums such as Pegasus, Apache Giraph, GraphLab [22], PBGL, etc. in future. We also plan to conduct performance evaluation of Dipper framework with graph sizes of billions of scale. Moreover, we plan to conduct more rigorous comparison of our approach with existing other graph manipulation platforms.

8. ACKNOWLEDGMENTS

This research was supported by the Japan Science and Technology Agency's CREST project titled "Development of System Software Technologies for post-Peta Scale High Performance Computing".

9. REFERENCES

[1] A. Buluç and J. R. Gilbert. The combinatorial blas: design, implementation, and applications. *International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[2] C. Bunch, B. Drawert, N. Chohan, C. Krintz, L. Petzold, and K. Shams. Language and runtime support for automatic configuration and deployment of scientific computing software over cloud fabrics. *Journal of Grid Computing*, 10:23–46, 2012.

[3] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, Aug. 2007.

[4] P. Charles et al. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[5] U. Consortium. *UPC Language Specifications, v1.2*. Lawrence Berkeley National Lab Tech Report LBNL-59208, 2005., 2011.

[6] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.

[7] S. Das et al. Ricardo: integrating r and hadoop. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 987–998, New York, NY, USA, 2010. ACM.

[8] T. Davis and Y. Hu. The university of florida sparse matrix collection. URL: `http://www.cise.ufl.edu/research/sparse/matrices/`, 2012.

[9] M. Dayarathna, C. Houngkaew, H. Ogata, and T. Suzumura. Scalable performance of scalegraph for large scale graph analysis. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–9, 2012.

[10] M. Dayarathna, C. Houngkaew, and T. Suzumura. Introducing scalegraph: an x10 library for billion scale graph analytics. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, pages 6:1–6:9, New York, NY, USA, 2012. ACM.

[11] J. Dongarra and et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[12] T. El-Ghazawi and L. Smith. Upc: unified parallel c. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[13] T. A. S. Foundation. Apache incubator giraph. URL: `http://incubator.apache.org/giraph/`, 2012.

[14] D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. *SIGPLAN Not.*, 40:423–437, October 2005.

[15] Gremlin. Gremlin. URL: `https://github.com/tinkerpop/gremlin/wiki/`, May 2012.

[16] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 349–362, New York, NY, USA, 2012. ACM.

[17] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a scalable and general graph management system. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '11, pages 1091–1099, New York, NY, USA, 2011. ACM.

[18] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society.

[19] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(2):387 –408, feb. 2005.

[20] E. Krepska, T. Kielmann, W. Fokkink, and H. Bal. Hipg: parallel processing of large-scale graphs. *SIGOPS Oper. Syst. Rev.*, 45(2):3–13, July 2011.

[21] K. Liu and E. Terzi. Towards identity anonymization on graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 93–106, New York, NY, USA, 2008. ACM.

[22] Y. Low et al. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[23] A. Lugowski, D. M. Alber, A. Buluç, J. R. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A flexible open-source toolbox for scalable complex graph analysis. In *SDM*, pages 930–941, 2012.

[24] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[25] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.

[26] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web, 1999.

[27] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, 2011.

[28] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-900051-07-0.

[29] M. A. Rodriguez and P. Neubauer. Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology*, 36(6):35–41, 2010.

[30] M. A. Rodriguez and P. Neubauer. The graph traversal pattern. *CoRR*, abs/1004.1001, 2010.

[31] S. Seo, E. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 721 –726, 30 2010-dec. 3 2010.

[32] B. Shao, H. Wang, and Y. Xiao. Managing and mining large graphs: systems and implementations. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 589–592, New York, NY, USA, 2012. ACM.

[33] The igraph Project. The igraph library. URL: `http://igraph.sourceforge.net/`, 2013.

[34] W3C. Sparql query language for rdf. URL: `http://www.w3.org/TR/rdf-sparql-query/`, 2012.

[35] X. Wu, X. Ying, K. Liu, and L. Chen. A survey of privacy-preservation of graphs and social networks. In C. C. Aggarwal, H. Wang, and A. K. Elmagarmid, editors, *Managing and Mining Graph Data*, volume 40 of *The Kluwer International Series on Advances in Database Systems*, pages 421–453. Springer US, 2010.

[36] A. Yoo and I. Kaplan. Evaluating use of data flow systems for large graph analysis. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, pages 5:1–5:9. ACM, 2009.