High-Performance Analysis of Filtered Semantic Graphs [*]

Aydın Buluç[†], Armando Fox[+], John R. Gilbert[*], Shoaib Kamil[+†], Adam Lugowski[*†],
Leonid Oliker, Samuel Williams
Lawrence Berkeley National Laboratory, +University of California at Berkeley,
and ∗University of California at Santa Barbara
abuluc@lbl.gov, fox@cs.berkeley.edu, gilbert@cs.ucsb.edu, skamil@cs.berkeley.edu,
alugowski@cs.ucsb.edu, loliker@lbl.gov, swwilliams@lbl.gov

ABSTRACT

High performance is a crucial consideration when executing a complex analytic query on a massive semantic graph. In a semantic graph, vertices and edges carry "attributes" of various types. Analytic queries on semantic graphs typically depend on the values of these attributes; thus, the computation must either view the graph through a *filter* that passes only those individual vertices and edges of interest, or else must first materialize a subgraph or subgraphs consisting of only the vertices and edges of interest. The filtered approach is superior due to its generality, ease of use, and memory efficiency, but may carry a performance cost.

In the Knowledge Discovery Toolbox (KDT), a Python library for parallel graph computations, the user writes filters in a high-level language, but those filters result in relatively low performance due to the bottleneck of having to call into the Python interpreter for each edge. In this work, we use the Selective Embedded JIT Specialization (SEJITS) approach to automatically translate filters defined by programmers into a lower-level efficiency language, bypassing the upcall into Python. We evaluate our approach by comparing it with the high-performance C++ /MPI Combinatorial BLAS engine, and show that the productivity gained by using a high-level filtering language comes without sacrificing performance.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming, Parallel programming*

Keywords

Domain Specific Languages, Graph Analysis, SEJITS, KDT, High-performance computing

1. INTRODUCTION

In a semantic graph, edges and/or vertices are labeled with *attributes* that may represent a timestamp, a type of relationship, or a mode of communication. An analyst (i.e. a user of graph analytics) may want to run a complex workflow over a large graph, but wish to only use those graph edges whose attributes pass a filter defined by the analyst. For example, in a graph where vertices represent Twitter users and edges represent "following" or "retweeting" relationships, the analyst may want to search through vertices reachable from a particular user via the subgraph consisting only of "retweet" edges with timestamps earlier than a given date.

Filters raise performance issues for large-scale graph analysis. In many applications, running a filter across an entire graph data corpus to materialize the filtered graph as a new object for analysis can be prohibitively expensive and creates storage problems. Moreover, the time spent during materialization is typically not amortized by many graph queries because the user modifies the query (or just the filter) during interactive data analysis. The alternative is to filter edges and vertices "on the fly" during execution of the complex graph algorithm. A graph algorithms expert can implement an efficient on-the-fly filter as a set of primitive Combinatorial BLAS [2] operations coded in C++ , but filters written at the higher-productivity KDT [5] level, as graph operations in Python, incur a significant performance penalty.

Our solution to this challenge is to apply Selective Just-In-Time Specialization techniques from the SEJITS approach [3]. We define an embedded domain-specific language (DSL) that uses a subset of Python to define semantic graph filters, and use the SEJITS methodology to implement the translation necessary for filters written in that subset to run as efficiently as the low-level C++ code. As a result, we are able to demonstrate that SEJITS technology significantly accelerates Python graph analytics codes written in KDT and running on clusters and multicore CPUs.

2. KDT FILTERS IN PYTHON

In KDT, any graph algorithm can be performed with an edge filter. A filter is a unary predicate on an edge that returns true if the edge is to be considered, or false if it is to be ignored. The KDT user writes a filter predicate as a Python function or lambda expression of one input that returns a boolean value.

KDT filters are applied directly to the lowest-level KDT structures; all operations on these structures then respect the filter. This means that all algorithms automatically respect filters with no modifications required. Additional filters can be added and removed at any time, with filter predicates evaluated in the order they are added. KDT supports two approaches for filtering semantic graphs: ***materializing filters*** and ***on-the-fly filters***.

[*]This work was supported in part by NSF grant CNS-0709385, by DOE grant DE-AC02-05CH11231, and by grants from Intel Corporation and Microsoft Corporation.
[†]Corresponding authors.

When a ***materializing filter*** is placed on a graph (or matrix or vector), the entire graph is traversed and a copy is made that includes only the edges that pass the filter. This touches every element only once and potentially doubles the memory requirements. By contrast, every primitive operation (e.g. semiring scalar multiply) applies an ***on-the-fly filter*** to its inputs when it is called. No copy is made, rather every primitive operation accesses the graph through the filter and behaves as if the filtered-out edges were not present. Only the filter predicates are stored. Each filtered element may be touched multiple times, but, depending on algorithm, some may not be touched at all.

3. SEJITS AND FILTERS

In order to mitigate the slowdown caused by defining filter predicates in Python, which results in a serialized upcall into Python for each operation, we opt to instead use the Selective Embedded Just-In-Time Specialization (SEJITS) approach [3]. By defining an embedded DSL for KDT filters, and translating it to C++ , we avoid performance penalties while still allowing users the flexibility to specify filters in Python. We use the Asp[1] framework to implement our DSL.

In the usual KDT case, filters are written as simple Python functions. Since KDT uses Combinatorial BLAS at the low level to perform graph operations, each operation at the Combinatorial BLAS level must check to see whether the vertex or edge should be taken into account, requiring a per-vertex or per-edge upcall into Python.

We define an embedded domain specific language for filters, and allow users to write their filters in this DSL, expressed as a subset of Python with normal Python syntax. At instantiation, the filter source code is introspected to get the Abstract Syntax Tree (AST), and then is translated into low-level C++ . Subsequent applications of the filter use this low-level implementation, sidestepping the serialization and cost of upcalling into Python. Formal definition of our domain-specific language and several examples of filters written in Python can be found in our technical report [1].

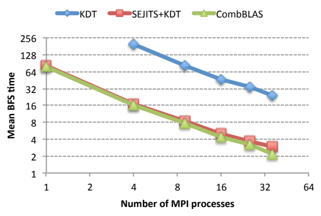
4. EXPERIMENTS

We use synthetically-generated R-MAT [4] graphs with a very skewed degree distribution. An R-MAT graph of scale N has 2^N vertices and approximately $edgefactor \cdot 2^N$ edges. Our *edgefactor* is 16, and R-MAT seed paratemeters a, b, c, and d are $0.59, 0.19, 0.19, 0.05$, respectively. After generating this non-semantic (boolean) graph, edge payloads are artificially introduced using a random number generator in a way that ensures target filter permeability. The edge type is a struct that is composed of a boolean isfollower flag, a timestamp, the number of retweets. Our technical report [1] contains experiments on graphs from real social network interactions on a distributed memory supercomputer.

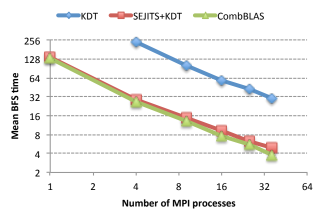
The parallel scaling of our approach and the relative performance of three methods is shown in Figure 1. The experiment is run on Mirasol, a single node platform composed of four Intel Xeon E7-8870 processors. The sustained stream bandwidth is about 30 GB/s per socket. Mirasol has 256 GB 1067 MHz DDR3 RAM. We use OpenMPI 1.4.3 with GCC C++ compiler version 4.4.5, and Python 2.6.6.

CombBLAS achieves remarkable linear scaling with increasing process counts (34-36X on 36 cores), while SEJITS+KDT closely tracks its performance and scaling. Single core KDT runs did not finish in a reasonable time to report. We do not report performance of materialized filters as they were the slowest by a large margin.

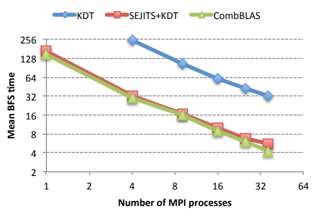
[1]https://github.com/shoaibkamil/asp/wiki/



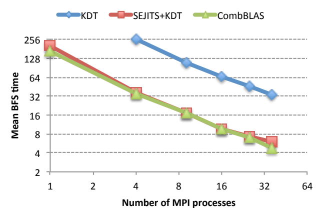
(a) 1% permeable



(b) 10% permeable



(c) 25% permeable



(d) 100% permeable

Figure 1: Parallel 'strong scaling' results of filtered BFS on Mirasol, with varying filter permeability on a synthetic data set (R-MAT scale 23). Both axes are in log-scale, time is in seconds.

5. CONCLUSION

The KDT graph analytics system achieves customizability through user-defined filters, high performance through the use of a scalable parallel library, and conceptual simplicity through appropriate graph abstractions expressed in a high-level language. We showed that the performance hit of expressing filters in a high-level language can be mitigated by Just-in-Time Specialization. In particular, our embedded DSL for filters enables Python code to achieve comparable performance to a pure C++ implementation.

6. REFERENCES

[1] A. Buluç, A. Fox, J. Gilbert, S. Kamil, A. Lugowski, L. Oliker, and S. Williams. High-performance analysis of filtered semantic graphs. Technical Report UCB/EECS-2012-61, May 2012.

[2] A. Buluç and J.R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications (IJHPCA)*, 25(4):496–509, 2011.

[3] B. Catanzaro, S.A. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K.A. Yelick, and A. Fox. SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization. In *PMEA*, 2009.

[4] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos. Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication. In *PKDD*, pages 133–145, 2005.

[5] A. Lugowski, D. Alber, A. Buluç, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis. In *SDM'12*, pages 930–941, April 2012.