SCALABLE COMPLEX GRAPH ANALYSIS WITH THE KNOWLEDGE DISCOVERY TOOLBOX

Adam Lugowski[†] *Aydın Buluç*[⋆] *John R. Gilbert*[†] *Steve Reinhardt*[‡]

[⋆] Lawrence Berkeley National Laboratory
[†]University of California, Santa Barbara
[‡]Microsoft Corporation

ABSTRACT

The Knowledge Discovery Toolbox (KDT) enables domain experts to perform complex analyses of huge datasets on supercomputers using a high-level language without grappling with the difficulties of writing parallel code, calling parallel libraries, or becoming a graph expert. KDT delivers competitive performance from a general-purpose, reusable library for graphs on the order of 10 billion edges and greater. We describe our approach for supporting arbirary vertex and edge attributes, in-place graph filtering, and graph traversal using pre-defined access patterns.

***Index Terms*—** graph analytics, scalability, knowledge discovery, semantic graph, filter

1. INTRODUCTION

Analysis of very large graphs has become indispensable in fields ranging from genomics and biomedicine to financial services, marketing, and national security, among others. Our Knowledge Discovery Toolbox (KDT) [1] is the first package that combines ease of use for these domain experts, scalability on supercomputers (large HPC clusters) where many domain scientists run their large scale experiments, and extensibility for graph algorithm developers. KDT addresses the needs both of graph analytics users (who are not expert in algorithms or high-performance computing) and of graph analytics researchers (who are developing algorithms and/or tools for graph analysis). KDT is an open-source, flexible, reusable infrastructure that implements a set of key graph operations with excellent performance on standard computing hardware.

KDT specifically targets those who are not experts in graph analytics. We believe allowing domain experts to directly explore graphs is necessary for discovery. Our goal is to build a package that a) is conceptually simple enough for domain experts to use, b) is customizable enough to solve their graph problems, and c) performs well enough to execute their graph problems in acceptable time and memory. With the authors' background in distributed-memory combinatorial supercomputing, we chose to start from a known performance base (Combinatorial BLAS [2]) and address the dimensions of conceptual simplicity (for domain experts) and customizability. It is unknown whether an implementation can be realized that meets all three requirements.

This paper describes new features of KDT that support graphs with attributes on both edges and vertices, so-called *semantic* graphs, and how those changes meet the criteria of customizability and performance.

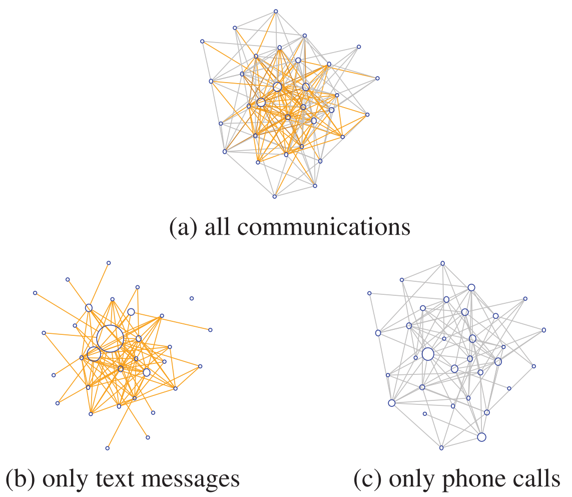

Fig. 1. Example of placing a filter on a graph. We compute betweenness centrality on a graph of communications consisting of both text messages and cell phone calls, then filter to only text messages or cell phone calls. A vertex's size indicates its normalized centrality score. Each filtered graph highlights different central nodes, leading to better understanding of communication patterns.

2. SEMANTIC GRAPH EXAMPLE

Consider the example of a social network where information is known about cell-phone calls and text messages. To understand the patterns of communication in the social network, an analyst may want to explore the graph by looking at

```
# the variable bigG contains the graph
# define the edge selection filter
def eFilter(self):
   return self.eType == eType

# for each edge type, calculate
# betweenness centrality
mList=(PhoneCall,TextMessage)
bigG.addEFilter(eFilter)
for eType in mList:
   bc = bigG.rank('approxBC')
   #visualize vertex centrality in graph composed of edges
       of only a single type

bigG.delEFilter(eFilter)
bc = bigG.rank('approxBC')
#visualize vertex centrality based on all edges
```

Fig. 2. KDT code implementing the semantic-graph example described in Section 2. All filtering is done dynamically without creating any intermediaries.

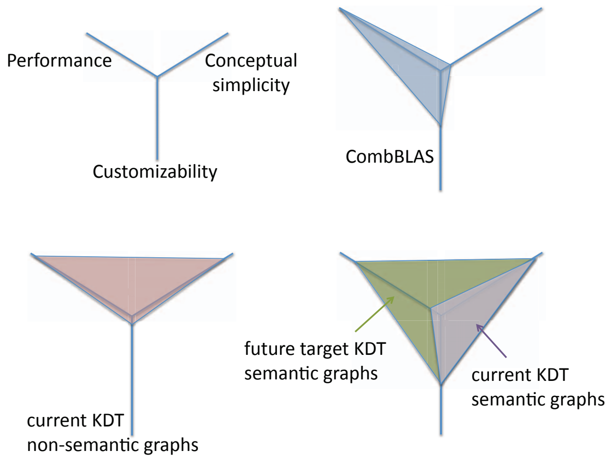
Performance
Conceptual simplicity
Customizability
CombBLAS
future target KDT semantic graphs
current KDT semantic graphs
current KDT non-semantic graphs

Fig. 3. A high-level comparison of advances in CombBLAS and KDT. Our current semantic graph implementation has high simplicity and customizability. Our target is to build on that by adding the performance of our current non-semantic graphs.

each mode of communication separately, with any of the algorithms supported in KDT. For example, betweenness centrality [3] often gives insight into those people (vertices) who most connect the whole graph. Calculating betweenness centrality considering only phone calls, and then only text messages may give deeper insight than calculating betweenness centrality considering both communication modes simultaneously. Note that the latter is not simply a linear combination of the former two. Figure 1 provides an illustration. This can be implemented in KDT v0.2 with the code in Figure 2.

An important aspect of this example is that the filtered graphs (*i.e.* the graph of only text messages) are never materialized. The predicates used to filter the edges are applied on-the-fly, thus eliminating the need to create intermediaries. The edge filter predicate `eFilter` is attached to the graph by the `addEFilter` method, and then executed whenever edge traversing operations are invoked.

This example has analogues in life sciences, where the different edges might be protein-protein or protein-DNA interactions.

3. KDT DESIGN

We build on our previous work on the Combinatorial BLAS (or CombBLAS for short) [2] by utilizing it as our initial backend. The CombBLAS is a proposed standard for combinatorial computational kernels. It is a highly-templated C++ library. It offers a small set of linear algebraic kernels that can be used as building blocks for the most common graph-analytic algorithms. Graph abstractions can be built on top of its sparse matrices, taking advantage of its existing best practices for handling parallelism in sparse linear algebra. Its flexibility comes from the arbitrary operations that it supports. The user, or in this case the KDT implementor, specifies the `add` and `multiply` routines in matrix-matrix and matrix-vector operations, or unary and binary functions for element-wise operations. The main data structures are distributed sparse matrices and vectors, which are distributed in a two-dimensional processor grid for scalability.

KDT transforms the linear algebra primitives into graph primitives. The graph's edges are collectively stored in a matrix, and vertex attributes are stored in a vector. Sparse matrix-vector multiplication (SpMV) and sparse matrix-matrix multiplication (SpGEMM) become KDT's graph traversal primitives, where user code in the `add` and `multiply` semiring routines defines the function of the traversal. Element-wise operations become edge and vertex visitors. The main benefit of this approach is that traditional graph frameworks are latency-bound whereas linear algebra primitives are bandwidth bound. The latter is far more scalable.

Our first KDT release focused on providing key abstractions on data structures and algorithms (*e.g.* digraphs, rank, cluster) and the supporting infrastructure (vectors, matrices, Python bindings). Our goal was to be able to deliver our world-class CombBLAS performance with conceptual simplicity and user-friendly design. We did not focus on extending the graph abstractions; instead we supported only floating-point attributes on both vertices and edges.

The progression of capabilities of CombBLAS and KDT is illustrated in Figure 3.

4. CUSTOMIZABILITY: SUPPORTING ATTRIBUTES FOR VERTICES AND EDGES

4.1. Datatypes

The primary feedback we received from potential KDT users on our initial release was the need to support semantic graphs, *i.e.*, graphs whose edges and vertices have attributes on them. The needed support consisted of two primary changes to KDT: the ability to create graphs with edge objects more complex than the single 64-bit data element of our first release (and similarly vectors with vertex objects more complex than the 64-bit element), and the ability to customize KDT operations to filter or compute on elements of the edge and vertex objects. These changes must be made balanced with the conceptual simplicity and performance requirements.

Our filter design relies on three basic principles.

1. A user-defined predicate determines whether or not a vertex or edge exists in the filtered graph

2. Multiple user-defined predicates can be stacked and the filters they define are applied in the order they are added to the graph. Thus, both users and algorithm developers can use filters.

3. All graph operations respect the filter. This ensures that algorithms can be written without taking filters into consideration at all, thus greatly easing their design.

Two performance issues constrain the semantic-graph design in KDT. First, KDT is targeted at complex graph analytics, which usually traverse the graph more than simple analytics. These traversals are time-consuming, so to avoid a catastrophic performance decrease when using semantic graphs in KDT, the semantic-graph mechanisms must support computations that require only minimally (and ideally no) more passes over the graph than the non-semantic case. Second, because of the traversal-intensive nature of complex graph analytics and the fact that in-memory operation is typically much faster than on-disk operation, frugal memory use will enable much larger problems to be solved. Specifically, when a user filters a graph to operate on only certain types of edges or vertices, avoiding the materialization of the intermediate graph will typically be a large saving in memory consumption. KDT's semantic-graph mechanisms strive to achieve this.

Given that KDT interfaces are via Python, a natural target for customizable data structures would be a fully general Python object. Unfortunately, Python objects are so general that even their size might not remain constant during their lifetime. KDT's dependence on the Combinatorial BLAS, a C++ package, requires a set of statically-typed and statically-sized objects known at compile time, which does not lend itself to straightforward support of general run-time definable Python objects. In practice, less-general structures targeted at semantic graphs provide the support needed for many semantic-graph problems; *e.g.*, STINGER [4] has been proposed as a common graph data structure.

We are continually relaxing our requirements for what an attribute can be. Our original implementation used simple 64-bit floating point scalar values as the only supported attribute types.

Currently we provide two statically-defined object types, `Obj1` and `Obj2`, which are motivated by STINGER. Unlike STINGER, however, our users may modify the object types, albeit in C++ at KDT compile time. Each object type, as well as scalars, can be used for either edge or vertex attributes. With this data-structure flexibility comes some additional user responsibility in defining how the elements of the objects are used, *i.e.*, how the `load` function will fill the members of the object from data values in an input file, overload operators if desired, etc.

We plan to eventually support arbitrary object types defined by the user in Python. These objects would be subject to the restriction that they do not change structure (size or makeup) during execution and that all elements of a matrix or vector (i.e. any particular graph) must have all attributes of the same type. These restrictions allow us to keep our high-performance communication methods, and are common in high-performance Python packages.

4.2. Computation

Computations on the edge and vertex objects consist of three types: *semirings* that perform the elemental calculation that occurs at each position of a dot product corresponding to a single step in a graph traversal (such as + or `min`), *element-wise* functions that define the behavior of elemental operations on edges or vertices, and *filter predicates* that return a Boolean True value for each vertex or edge to be retained in the computation.

KDT's breadth-first search function is an illustrative example. For a graph with no attributes, at each step the *fringe* vertices that were newly encountered on the previous step have their out-edges examined. If a previously unvisited vertex is encountered, the source vertex of the edge to the new vertex is remembered as the *parent* (in case of multiple edges from fringe vertices to the new vertex, the highest-numbered source vertex is remembered).

The semiring multiply operation visits an edge; the add operation consolidates multiple edges coming into a single vertex (using a *max* operation in our example). Element-wise operations are used to determine if a vertex is newly discovered, for updates to the parents, and for pruning the frontier of discovered vertices.

Applying a filter to either the edges or vertices effectively removes the filtered elements from the graph. For example, a user may want to calculate a time-dependent path operation for just CellPhone edges, and the time-dependent operation itself may filter edges based on their start times. A detailed explanation of how our filtering system works follows in Section 4.3.

4.3. In-place graph filtering

In addition to the three filter principles listed in Section 4.1, we take the step of implementing filters at a high level. Our backend can thus be designed without explicit support for filtering, greatly simplifying its implementation. Our backend supports operations that fall into three basic categories. We have element-wise operations of the form $e_i = f(e_i)$, operations to select elements based on a predicate (*eg.* Count), and semiring operations (SpMV, SpGEMM). Each operation lends itself to supporting filters without altering its basic implementation.

The element-wise operations can be filtered by introducing a shim function $s(x)$ which traverses the filter predicate stack and determines if the element x is kept or not. If not, $s(x)$ returns x and the result is a no-op. If x passes the filter then the user's operation called and $s(x)$ returns $f(x)$.

A similar shim is used for the predicate operations. The filter stack essentially adds additional logical AND terms to the predicate.

SpMV and SpGEMM operations using semirings are both filtered in the multiply step. If either element is filtered out then the multiply becomes a no-op, as if it didn't happen at all. The SpGEMM case can again be implemented with a simple shim in the multiply operation. The SpMV case is more complex because of the semantics of the vector's filter. A filter on the vector means that vertices of the graph are filtered. If a vertex is filtered out then all edges incident to it must also be filtered out. In the SpMV data pattern, the multiply operation only has the values of vertices at the tails of the edges, but not the heads. A naïve application of the vertex filter would not filter out edges whose heads are incident to a vertex which is filtered out. A solution is to turn the vertex filter into an edge filter by adding a boolean flag to each edge. The vertex filter is applied once to the vector, and its result is broadcast along the rows and columns of the matrix. The SpMV's multiply operation can now filter on just an edge filter.

5. PERFORMANCE

A key performance aspect is the ability to run user code efficiently in the most inner loops of the framework. The ideal solution is to efficiently execute code written by the user in the high level language (Python). This, however, introduces the performance penalty of calling into an interpreter for every operation.

An alternative solution is to pre-define a set of composable primitives which are implemented in the fast low-level language but exposed in the high level one. The user then composes their operation from these primitives. We found this approach to provide near hard-coded speed and approximately 80X performance benefits over calling Python code because the callback into the interpreter is eliminated. The price is reduced ease of use.

A superior approach is to run code written in Python at C speeds. This is the goal of SEJITS [5], which provides a translation and compilation framework for Python which automatically accelerates repeated operations. It translates the operation to C++, compiles it, then calls the native code instead of the original Python code. The heavy-lifting task of optimization is left to the C++ compiler so the SEJITS framework itself is very lightweight. In current work we are exploring the use of SEJITS to accelerate KDT.

6. CONCLUSION

We have introduced KDT, an easy to use, high performance graph computation library. We demonstrated KDT's increasing flexibility in the types of graphs it can represent and operations it supports. Namely we described arbitrary attributes on vertices and edges, and custom user-defined operations for writing graph algorithms using high-performance patterns. We also introduced the ability to filter graphs in-place without incurring additional storage requirements. We also showed that despite their customizability and user-friendliness, these operations can still be efficiently performed.

7. REFERENCES

[1] A. Lugowski, D. Alber, A. Buluç, J.R. Gilbert, S. Reinhardt, Yun Teng, and Andrew Waranis, "A flexible open-source toolbox for scalable complex graph analysis," in *SIAM Conference on Data Mining (SDM)*, 2012 (accepted).

[2] A. Buluç and J.R. Gilbert, "The Combinatorial BLAS: Design, Implementation, and Applications," *The International Journal of High Performance Computing Applications*, vol. online first, 2011.

[3] L.C. Freeman, "A Set of Measures of Centrality Based on Betweenness," *Sociometry*, vol. 40, no. 1, pp. 35–41, 1977.

[4] D.A. Bader, J. Berry, A. Amos-Binks, D. Chavarría-Miranda, C. Hastings, K. Madduri, and S.C. Poulos, "STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) extensible representation," *Georgia Institute of Technology, Tech. Rep*, 2009.

[5] 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," Tech. Rep. UCB/EECS-2010-23, EECS Department, University of California, Berkeley, Mar 2010.