Enabling Rapid Development and Execution of Advanced Graph-Analysis Algorithms on Very Large Graphs

Aydin Buluc, LBL (abuluc@lbl.gov)
John Gilbert and Adam Lugowski, UCSB ({gilbert,alugowski}@cs.ucsb.edu)
Steve Reinhardt, Microsoft (steve.reinhardt@microsoft.com)

With ideas from
Dave Wecker and Zheng Zhang, Microsoft Research
Jim Harrell, Cray, Inc.
Viral Shah, formerly UCSB
Knowledge Discovery Toolbox (KDT) embodies two key innovations:

- Technically, non-graph-expert subject-matter experts analyze terascale graphs with multiple advanced algorithms with leading performance

- Architecturally, graph algorithm users, graph algorithm developers, and graph infrastructure developers each use complementary interfaces to advance the field
Agenda

• APIs for different audiences
• Semantic and hyper-graphs
• Implementation / performance
Knowledge Discovery Workflow

1. Cull relevant data
2. Build input graph
3. Analyze input graph
4. Visualize result graph

- Gene
- Email
- Twitter
- Facebook
- Video
- Sensor
- Web
- ...

DryadLINQ, StreamInsight

KDT

??
Agenda

• APIs for different audiences
  • Semantic and hyper-graphs
  • Implementation / performance
KDT APIs enable disparate groups’ work to reinforce each other

- Fosters earlier use and learning about how algorithms work at scale

**Graph-algorithm users**
develop applications based on a set of complex graph algorithm implemented by experts

**Graph-algorithm developers**
develop algorithms for a growing set of users through an evolving set of interfaces, based on powerful infrastructure

**Graph-infrastructure developers**
develop new implementations of the KDT interfaces for different hardware or software platforms
KDT APIs enable disparate groups’ work to reinforce each other.

```python
# Graph500.py
deg3verts = (G.degree() > 2).findInds()
deg3verts.randPerm()
starts = deg3verts[kdt.ParVec.range(nstarts)]
G.toBool()
[origI, ign, ign2] = G.toParVec()
for start in starts:
    parents = G.bfsTree(start, sym=True)
    nedges = len((parents[origI] != -1).find())
    if not k2Validate(G, start, parents):
        verifyResult = "FAILED"
```

Graph-algorithm users

Graph-algorithm developers

Graph-infrastructure developers

Architecturally
KDT APIs enable disparate groups’ work to reinforce each other

Technically

Architecturally

Graph-algorithm users

Graph-algorithm developers

Graph-infrastructure developers

L = G.toSpParMat()
d = L.sum(kdt.SpParMat.Column)
L = -L
L.setDiag(d)
M = kdt.SpParMat.eye(G.nvert()) - mu*L
pos = kdt.ParVec.rand(G.nvert())
for i in range(nsteps):
    pos = M.SpMV(pos)
KDT APIs enable disparate groups’ work to reinforce each other

```python
# community detection due to Botherel and Bouklit
import kdtxmt

import kdtxmt

Q = kdt.ParVec.zeros(G.nedge())
for i in range(G.nedge()):
    bc = kdtxmt.centrality(G, 'approxBC', 'edge')
    G.delete_edge(bc.maxndx()[1])
    p = G.cluster()
    Q[i] = G.modularity(p)
best = Q.max()
```

// SWIG headers for kdtxmt.py
```
// SWIG headers for kdtxmt.py

#include "pyCentrality.h"
```

Graph-algorithm users
Graph-algorithm developers
Graph-infrastructure developers

Technically
Architecturally
KDT’s Graph API (v0.1)

- Targeted at non-graph-expert domain experts
- Exposed via Python

Real applications

Community Detection

Network Vulnerability Analysis

Applets

- centrality('exactBC')
- centrality('approxBC')
- Graph500
- pageRank

Building blocks

- DiGraph
  - bfsTree, isBfsTree
  - plus utility (e.g., DiGraph,nvert, toParVec,degree,load,UFget,+,*,sum,subgraph,reverseEdges)

(Sp)ParVec
  - (e.g., +,*,[],&,,>,==,abs,max,sum,range,norm,randPerm,scale,topK)

CombBLAS
  - SpMV_SemiRing,
  - SpMM_SemiRing
KDT’s Graph API (v0.2)

Real applications

Community Detection

Network Vulnerability Analysis

Applets

centrality(‘exactBC’)
centrality(‘approxBC’)

Graph500

pageRank

cluster(‘Markov’)
cluster(‘spectral’)

Building blocks

DiGraph
bfsTree, isBfsTree
plus utility (e.g., DiGraph,nvert, toParVec,degree,load,UFget,+,*,
sum,subgraph,reverseEdges)

HyGraph
bfsTree, isBfsTree
plus utility (e.g., HyGraph,nvert, toParVec,degree,load,UFget)

SpParMat
(e.g., +,* , SpMM, SpMV,
SpMM_SemiRing,
SpMV_SemiRing,
SpParMat
(e.g., +,* , SpMM, SpMV,
SpMM_SemiRing,
SpMV_SemiRing, SpMM_SemiRing)

(Sp)ParVec
(e.g., +,* , |,&,>,==,[]
abs,max,sum,range, norm, randPerm, scale, topK)

ComblAS

SpMV_SemiRing,
SpMM_SemiRing
Agenda

• APIs for different audiences

• Semantic and hyper-graphs

• Implementation / performance
Semantic-graph API: Multiple Criteria

- **Technically**
- **Architecturally**

**Level of abstraction**
- Customizability
  - Performance

**Performance**
- CombBLAS
  - Atypical abstractions
  - Sustainably scalable performance
- PBGL
  - Abstractions low-level for domain experts
  - Scalable performance
- KDT v0.2 goal
Semantic Graph Use Case

- **Vertex types:** Person, Smartphone, Camera
- **Edge types:** PhoneCall, TextMessage, PhysicalPresence
- **Edge StartTime, EndTime:**
  - Calculate betweenness centrality just for PhoneCalls and TextMessages between People occurring between times sTime and eTime
Approach 1: Known Good Performance

```
def vfilter(self, wantedVTypes):
    return kdt.in(wantedVTypes, self.type)

def efilter(self, wantedETypes, sTime, eTime):
    return kdt.and(kdt.in(wantedETypes, self.type),
                   kdt.and(kdt.gt(sTime, self.sTime),
                           kdt.lt(eTime, self.eTime)))

wantedVTypes = (People)
wantedETypes = (PhoneCall, TextMessage)
bc = Gtmp.centrality('approxBC', filter=(vfilter, efilter))
```
Approach 2: Highly Flexible, Currently Bad Performance

def vfilter(self, wantedVTypes):
    # any Python constructs permitted
    return self.type in wantedVTypes

def efilter(self, wantedETypes, sTime, eTime):
    return (self.type in wantedETypes)
    and (sTime > self.sTime)
    and (eTime < self.eTime)

wantedVTypes = (People)
wantedETypes = (PhoneCall, TextMessage)
bc = G.centrality('approxBC', filter=(vfilter, efilter))
Approach 3: Likely Good Performance, but Potentially Memory-Expensive

```
def vfilter(self, wantedVTypes):
    return self.type in wantedVTypes

def efilter(self, wantedETypes, sTime, eTime):
    return (self.type in wantedETypes)
    and (sTime > self.sTime)
    and (eTime < self.eTime)

wantedVTypes = (People)
wantedETypes = (PhoneCall, TextMessage)
Gtmp = G.subgraph(filter=(vfilter, efilter))
bc = Gtmp.centrality('approxBC')
```
Hypergraph Support

- The underlying sparse matrix is interpreted as an incidence matrix; vertices are in columns, edges in rows
- (Subset of) same methods implemented
- Graph500 Kernel 2 looks identical except validation
- Performance not yet measured for big cases, but expected to take twice as long as same DiGraph method
  - Two SpMVs in the core loop instead of one
  - TEPS rating the same
def bfsTree(self, root, sym=False):
    if not sym:
        self._T()
    parents = pcb.pyDenseParVec(self.nvert(), -1)
    fringe = pcb.pySpParVec(self.nvert())
    parents[root] = root
    fringe[root] = root
    while fringe.getnee() > 0:
        fringe.setNumToInd()
        self._spm.SpMV_SelMax_inplace(fringe)
        pcb.EWiseMult_inplacefirst(fringe, parents, True, -1)
        parents[fringe] = 0
        parents += fringe
    if not sym:
        self._T()
    return ParVec.toParVec(parents)

def bfsTree(self, root):
    parents = pcb.pyDenseParVec(self.nvert(), -1)
    fringeV = pcb.pySpParVec(self.nvert())
    parents[root] = root
    fringeV[root] = root
    while fringeV.getnee() > 0:
        fringeV.setNumToInd()
        fringeE = self._spm.SpMV_SelMax(fringeV)
        fringeV = self._spmT.SpMV_SelMax(fringeE)
        pcb.EWiseMult_inplacefirst(fringeV, parents, True, -1)
        parents[fringeV] = 0
        parents += fringeV
    return ParVec.toParVec(parents)
Questions about Hypergraph Support

• We have defined a BFS tree of a hypergraph as a set of simple edges, each contained in a hyperedge (which permits cycles of hyperedges). Is this the most useful definition?

• Are hypergraphs in the KDT style useful? What use cases should we target? What methods should we provide?
Agenda

- APIs for different audiences
- Semantic and hyper-graphs
- Implementation / performance
Key DiGraph Methods in KDT v0.1/v0.2

def pageRank(self, epsilon=0.1, dampingFactor=0.85):
def centrality(self, alg, **kwargs):
    'exactBC', normalize=True
    'approxBC', sample=0.05, normalize=True
def cluster(self, alg, **kwargs):
    'Markov'
    'spectral'
def bfsTree(self, root, sym=False):
def isBfsTree(self, root, parents, sym=False):
def neighbors(self, source, nhop=1, sym=False):
def pathsHop(self, source, sym=False):
def degree(self, dir=gr.Out):
def genGraph500Edges(self, scale):
def load(fname):
def UFget(fname):
def max(self, dir):
def reverseEdges(self):
def scale(self, other, dir=gr.Out):
def sum(self, dir):
def DiGraph(sourceV, destV, weight, nvert):
def toParVec(self):
def toBool(self):
def normalizeEdgeWeights(self):
class Graph: #base class only
class DiGraph:
class ParVec:
class SpParVec:
class SpParMat:
def sendFeedback(): # may want to disable this
Key HyGraph Methods in KDT v0.2

def pageRank(self, epsilon=0.1, dampingFactor=0.85):

def centrality(self, alg, **kwargs):
    ‘exactBC’, normalize=True
    ‘approxBC’, sample=0.05, normalize=True

def cluster(self, alg, **kwargs):

def bfsTree(self, root, sym=False):

def isBfsTree(self, root, parents):

def neighbors(self, source, nhop=1):

def pathsHop(self, source):

def degree(self, dir=gr.Out):

def genGraph500Edges(self, scale):

def load(fname):

def UFget(fname):

def max(self, dir):

def invertEdgesVertices(self):

def scale(self, other, dir=gr.Out):

def sum(self, dir):

def HyGraph(edgeNumV, incidentVertexV, weightV, nvert):

def toParVec(self):

def toBool(self):

def toDiGraph(self):

def normalizeEdgeWeights(self):
Graph500 Performance [Aydin Buluc]

- Excellent scaling up to 2500 cores, good to 5K cores
  - LBL/NERSC’s Hopper Cray XE6
- Scale 29 (“mini”) has 8B directed edges
- Performance measured from Python

- On-node thread parallelism starts to show benefit at 10K cores and above
KDT development and licensing

• KDT is a collaboration among UCSB (John Gilbert et al), LBL (Aydin Buluc), and Microsoft Technical Computing
• The resulting software is released under the New BSD license
• v0.1 was released on March 17
  • Tested on Linux x86 and Cray XT configurations
• V0.2 release targeted for early June
• The project homepage is kdt.sourceforge.net
  • Downloads, User Guide, FAQ and bug reporting
Planned KDT v0.2 Content

• Windows HPC Server version
• Semantic graphs
• Hypergraphs
• Clustering - Markov and spectral

• Out-of-core (Dryad-based) version (likely v0.3)
• Cray XMT version
  – Discussing with Cray et al.
• Version based on other graph infrastructures
  – E.g., Parallel Boost Graph Library, SNAP, MultiThreaded Graph Library
Knowledge Discovery Toolbox (KDT) embodies two key innovations:

- Technically, non-graph-expert subject-matter experts analyze terascale graphs with multiple advanced algorithms with leading performance
- Architecturally, graph algorithm users, graph algorithm developers, and graph infrastructure developers each use complementary interfaces to advance the field
Backup
Graphs-on-Disk Use Case

Does graph analysis make sense on data that won’t all fit in memory?
Graphs-on-Disk Use Case

Does graph analysis make sense on data that won’t all fit in memory?

- The sparse-matrix-linear-algebra approach structures communication, so raw pointer-chasing performance not so important
- People are building sparse-matrix packages on top of MapReduce/Hadoop
- We will shortly map the KDT APIs onto a sparse-matrix package based on Dryad*

```python
import kdtooc
[
...
]  
G = kdtooc.load('mydata')
G.bfsTree(...)
```

Questions about KDT-on-disk Support

• Assuming that in-memory processing is much faster than on-disk (10X?), what type of graph ops would be practical for on-disk data? Just simple ops? Would something as compute-intensive as BC ever make sense out-of-core?

• Is semantic graph’s filtering capability essential for on-disk processing?
KDT Implementation on Combinatorial BLAS

- Combinatorial BLAS
  - Built for combinatorial (sparse-matrix) problems
    - Not limited to simple directed graphs
  - Powers the functionality and performance of KDT
  - Scales well to 2K-4K cores

Real applications

Community Detection

Network Vulnerability Analysis

Applets

- centrality('exactBC')
- centrality('approxBC')
- bfsTree, isBfsTree, neighbors, pathsHop
- Graph500
- pageRank
- DiGraph utility
  - (e.g., DiGraph (from edges), nverts, degrees, +, *, toParVec, subgraph, reverseEdges, load)
- ParVec/SpParVec utility
  - (e.g., +, -, *, |, &,, >, ==, [], abs, range, max, sum, norm, randPerm, topK)

Building blocks

- SpMV_SemiRing, SpMM_SemiRing
- Sparse-matrix classes/ops/types
  - (e.g., Apply, EWiseApply, Reduce)
Example Implementation: bfsTree

Technically
Ecologically

Implement:

\[ A^T \]
from

A^T

X

A^{TX}
Technically

Ecologically

from

\[ A^T X \]

to
Technically

Ecologically

from

to

\[ \begin{pmatrix} 
\begin{array}{ccc}
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot 
\end{array}
\end{pmatrix}
\begin{pmatrix} 
\begin{array}{c}
1 \\
7 
\end{array}
\end{pmatrix}
\Rightarrow
\begin{pmatrix} 
\begin{array}{c}
A^T \\
X \\
A^{TX}
\end{array}
\end{pmatrix} \]
bfsTree Implementation in KDT, for DiGraphs
(Kernel 2 of Graph500)

```python
def bfsTree(self, root, sym=False):
    if not sym:
        self.T()  # synonym for reverseEdges
    parents = dg.ParVec(self.nvert(), -1)
    fringe = dg.SpParVec(self.nvert())
    parents[root] = root
    fringe[root] = root
    while fringe.nnn() > 0:
        fringe.spRange()
        self._spm.SpMV_SelMax_inplace(fringe._spv)
        pcb.EWiseMult_inplacefirst(fringe._spv,
                                   parents._dpv, True, -1)
        parents[fringe] = fringe
    if not sym:
        self.T()
    return parents
```

- SpMV and EWiseMult are CombBLAS ops that do not yet have good graph abstractions
  - pathsHop is an attempt for one flavor of SpMV
def pageRank(self, epsilon = 0.1, dampingFactor = 0.85):
    # We don't want to modify the user's graph.
    G = self.copy()
    nvert = G.nvert()

    G._spm.removeSelfLoops()

    # Handle sink nodes (nodes with no outgoing edges) by
    # connecting them to all other nodes.
    degout = G.degree(gr.Out)
    nonSinkNodes = degout.findInds()
    nSinkNodes = nvert - len(nonSinkNodes)
    iInd = ParVec(nSinkNodes*(nvert))
    jInd = ParVec(nSinkNodes*(nvert))
    wInd = ParVec(nSinkNodes*(nvert), 1)
    sinkSuppInd = 0

    for ind in range(nvert):
        if degout[ind] == 0:
            # Connect to all nodes.
            for sInd in range(nvert):
                iInd[sinkSuppInd] = sInd
                jInd[sinkSuppInd] = ind
                sinkSuppInd = sinkSuppInd + 1

    sinkMat = pcb.pySpParMat(nvert, nvert,
                              iInd._dpv, jInd._dpv, wInd._dpv)

    sinkG = DiGraph()
    sinkG._spm = sinkMat

• This portion looks more like graph operations
pageRank Implementation in KDT (p. 2 of 2)
(main loop)

```
G.normalizeEdgeWeights()
sinkG.normalizeEdgeWeights()

# PageRank loop
delta = 1
dv1 = ParVec(nvert, 1./nvert)
v1 = dv1.toSpParVec()
prevV = SpParVec(nvert)
d dampingVec = SpParVec.ones(nvert) *
    ((1 - dampingFactor)/nvert)
while delta > epsilon:
    prevV = v1.copy()
    v2 = G._spm.SpMV_PlusTimes(v1._spv) + \n        sinkG._spm.SpMV_PlusTimes(v1._spv)
    v1._spv = v2
    v1 = v1*dampingFactor + dampingVec
    delta = (v1 - prevV)._spv.Reduce(pcb.plus(),
        pcb.abs())
return v1
```

• This portion looks much more like matrix algebra
Graph500 Implementation in KDT (p. 1 of 2)

scale = 15
nstarts = 640

GRAPH500 = 1
if GRAPH500 == 1:
    G = dg.DiGraph()
    K1elapsed = G.genGraph500Edges(scale)
    if nstarts > G.nvert():
        nstarts = G.nvert()
    deg3verts = (G.degree() > 2).findInds()
    deg3verts.randPerm()
    starts = deg3verts[dg.ParVec.range(nstarts)]
G.toBool()

K2elapsed = 1e-12
K2edges = 0
for start in starts:
    start = int(start)
    if start==0:    #HACK:  avoid root==0 bugs for now
        continue
    before = time.time()
    parents = G.bfsTree(start, sym=True)
    K2elapsed += time.time() - before
    if not k2Validate(G, start, parents):
        print "Invalid BFS tree generated by bfsTree"
        print G, parents
        break
    [origI, origJ, ign] = G.toParVec()
    K2edges += len((parents[origI] != -1).find())
Graph500 Implementation in KDT (p. 2 of 2)

```python
def k2Validate(G, start, parents):
    ret = True
    bfsRet = G.isBfsTree(start, parents)
    if type(ret) != tuple:
        if dg.master():
            print "isBfsTree detected failure of Graph500 test %d" % abs(ret)
        return False
    (valid, levels) = bfsRet

    # Spec test #3:
    [origI, origJ, ign] = G.toParVec()
    li = levels[origI]
    lj = levels[origJ]
    if not ((abs(li-lj) <= 1) | ((li==-1) & (lj==-1))).all():
        if dg.master():
            print "At least one graph edge has endpoints whose levels differ by more than one and is in the BFS tree"
        print li, lj
        ret = False

    # Spec test #4:
    neither_in = (li == -1) & (lj == -1)
    both_in = (li > -1) & (lj > -1)
    out2root = (li == -1) & (origJ == start)
    if not (neither_in | both_in | out2root).all():
        if dg.master():
            print "The tree does not span the connected component exactly, root=%d" % start
        ret = False

    # Spec test #5:
    respects = abs(li-lj) <= 1
    if not (neither_in | respects).all():
        if dg.master():
            print "At least one vertex and its parent are not joined by an original edge"
        ret = False

    return ret
```

- #1 and #2: implemented in isBfsTree

- #3: every input edge has vertices whose levels differ by no more than 1. Note: don't actually have input edges, will use the edges in the resulting graph as a proxy

- #4: the BFS tree spans a connected component's vertices (== all edges either have both endpoints in the tree or not in the tree, or source is not in tree and destination is the root)

- #5: a vertex and its parent are joined by an edge of the original graph
def isBfsTree(self, root, parents, sym=False):
    ret = 1  # assume valid
    nvertG = self.nvert()

    # calculate level in the tree for each vertex; root is at level 0
    if not sym:
        self.reverseEdges()
    parents2 = ParVec.zeros(nvertG) - 1
    parents2[root] = root
    fringe = SpParVec(nvertG)
    fringe[root] = root
    levels = ParVec.zeros(nvertG) - 1
    levels[root] = 0

    level = 1
    while fringe.nnn() > 0:
        fringe.spRange()
        # ToDo: create PCB graph-level op
        self._spm.SpMV_SelMax_inplace(fringe._spv)
        # ToDo: create PCB graph-level op
        pcb.EWiseMult_inplacefirst(fringe._spv, parents2._dpv, True, -1)
        parents2[fringe] = fringe
        levels[fringe] = level
        level += 1
    if not sym:
        self.reverseEdges()
isBfsTree implementation KDT (p. 2 of 2)

# build a new graph from just tree edges
tmp2 = parents != ParVec.range(nvertG)
treeEdges = (parents != -1) & tmp2
treeI = parents[treeEdges.findInds()]
treeJ = ParVec.range(nvertG)[treeEdges.findInds()]
if (treeJ == root).any():
    return -1
# note treeJ/TreeI reversed, so builtGT is transpose, as needed by SpMV
builtGT = DiGraph(treeJ, treeI, 1, nvertG)
visited = ParVec.zeros(nvertG)
visited[root] = 1
fringe = SpParVec(nvertG)
fringe[root] = root
cycle = False; multiparents = False
while fringe.nnn() > 0 and not cycle and not multiparents:
    fringe.spOnes()
    newfringe = SpParVec.toSpParVec(builtGT._spm.SpMV_PlusTimes(fringe._spv))
    if visited[newfringe.toParVec().findInds()].any():
        cycle = True
        break
    if (newfringe > 1).any():
        multiparents = True
    fringe = newfringe
    visited[fringe] = 1
if cycle or multiparents:
    return -1

# spec test #2
if (levels[treeI]-levels[treeJ] != -1).any():
    return -2

return (ret, levels)

- #1: validate that the tree is a tree and has no cycles:
  - a) no edge has the root as a destination
  - b) no cycle exists
  - c) no vertex has more than 1 parent

- #2: tree edges should be between vertices whose levels differ by 1