Extended Sparse Matrices as Tools for Graph Computation

Adam Lugowski

UCSB

SIAM Annual Meeting    July 11, 2012
Knowledge Discovery Toolbox

kdt.sourceforge.net

UCSB
BERKELEY LAB
The Parallel Computing Laboratory
Microsoft
NSF
Office of Science
U.S. Department of Energy
KDT Graphs: distributed sparse matrices

Edge attributes can be arbitrary objects

Transposed Adjacency Matrix:
spare structure distributed
in 2D layout
Graph Traversals are $M\times M$ or $M\times V$

User-defined semirings on user-defined objects

distance 1 from vertex 7
Algorithm logic in custom semirings

Semiring:

```python
def mul(x, y):
    return y

def add(x, y):
    return y
```

G

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

\[ f_{in} \times = f_{out} \]

```
f_{in} =

\[
\begin{align*}
\text{mul}(1, 3) & \quad \text{mul}(1, 5) \\
\text{add}(3, 5) &
\end{align*}
\]

f_{out} =

```
```

```
mul(1, 3) \\
mul(1, 5) \\
add(3, 5)
```
Sparse Matrix Operations

<table>
<thead>
<tr>
<th>Matrix-Matrix multiplication</th>
<th>Apply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix-Vector multiplication</td>
<td>Reduce</td>
</tr>
<tr>
<td>Element-Wise (eg. A .* B)</td>
<td>Prune</td>
</tr>
<tr>
<td>Scale by Vector</td>
<td>Find</td>
</tr>
</tbody>
</table>

All customizable with user-defined callbacks
Why (sparse) adjacency matrices?

<table>
<thead>
<tr>
<th>Traditional graph computations</th>
<th>Graphs in the language of linear algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data driven, unpredictable communication</td>
<td>Fixed communication patterns</td>
</tr>
<tr>
<td>Irregular and unstructured, poor locality of reference</td>
<td>Operations on matrix blocks exploit memory hierarchy</td>
</tr>
<tr>
<td>Fine grained data accesses, dominated by latency</td>
<td>Coarse grained parallelism, bandwidth limited</td>
</tr>
</tbody>
</table>
Complex methods
- centrality('approxBC')
- pageRank

Building blocks
- DiGraph
  - bfsTree, neighbor
  - degree
  - load, UFget
  - +, -, sum, scale
- Mat
  - reduce, scale
  - +, []
- Vec
  - max, norm, sort
  - sum, ceil
  - range, ones
  - +, -, *, /, >, ==, &, []

Underlying infrastructure (Combinatorial BLAS)
- SpMV, SpGEMM
- Classes/methods (eg, Apply, EWiseApply, Reduce)

Domain Experts
- Algorithm Experts
- HPC Experts
Example workflow
Example workflow KDT code

# the variable bigG contains the input graph
# find and select the giant component
comp = bigG.connComp()
giantComp = comp.hist().argmax()
G = bigG.subgraph(comp==giantComp)

# cluster the graph
clus = G.cluster('Markov')

# contract the clusters
smallG = G.contract(clus)
BFS on a Scale 29 RMAT graph
(500M vertices, 8B edges)

![Graph showing BFS performance with different number of cores]

Machine: NERSC’s Hopper
Ongoing work: 
High-performance Python

1. Speed up Python callbacks

1. Introducing runtime-defined types
Python is great at high-level operations, slow at inner loops.

*The way to make Python fast is to not use Python.*

---

**SEJITS** (A. Fox and S. Kamil)

- Selective Embedded Just-In-Time Specialization
  1. Take Python code
  2. Translate it to equivalent C++ code
  3. Compile with GCC
  4. Call compiled version instead of Python version

SEJITS: Speeding up Python with C++

SEJITS converts Python routine to C++

def mul(x, y):
    return y

double mul(const Obj2& arg1, double arg2) {
    return arg2;
}

Compiles it (gcc) at runtime.

Compiled C++ routine called instead of Python

mul(1, 3)      mul(1, 5)

3 4

mul.o
A filter is a predicate (Python function) which returns True if an edge is to be kept, False otherwise.
texts and phone calls

# draw graph
draw(G)

# Each edge has this attribute:
class edge_attr:
    isText
    isPhoneCall
    weight
Betweenness Centrality

```
bc = G.centrality("approxBC")
# draw graph with node sizes
# proportional to BC score
draw(G, bc)
```
Betweenness Centrality on texts

# BC only on text edges
G.addEFilter(
    lambda e: e.isText)
bc = G.centrality("approxBC")

# draw graph with node sizes
# proportional to BC score
draw(G, bc)
SEJITS brings performance back

Time (in seconds) for a single BFS iteration on Scale 23 RMAT (8M vertices, 130M edges) with 10% of elements passing filter. Machine is Mirasol.
Roofline analysis: why this works
Attributes

“Graph”

“Weighted Graph”

“Semantic Graph”
Extended Attribute Support

• Completely remove user-written C++ code
  – User friendliness, allows systemwide installs
• adds flexibility
  – remove limitations on number of types allowed
  – remove limitation on assumption of what an object is
  – allows definition of well-formatted datafiles
Extended Attribute Support

• Requirements:
  – Type defined in Python
    • Fixed-size
  – Memory allocated in C++, object used in Python
  – Be able to operate on Python-defined structure through C++
    • For SEJTIS

Regular Python objects too general
Extended Attribute Support

• Inspiration from ctypes.Structure:

class MyEdge(Structure):
    _fields_ = [("weight", c_double),
                ("isPhoneCall", c_bool),
                ("isText", c_bool)]
Acts like Python, C++ friendly

Python:

e = MyEdge()
e.weight = 10

But also have:

• sizeof, addressof, offset, type
• placement new

Can generate translations at runtime, performance equivalent to compile time-defined structs
Conclusion

• KDT is a high-performance graph analysis toolkit written for a high-productivity language
• Possible to write callbacks in high-level language while retaining low-level language performance
• Possible to define datatypes at runtime
Thank You

kdt.sourceforge.net