A sustainable software stack for parallel graph analysis

Aydın Buluç
Lawrence Berkeley National Laboratory
July 11, 2012
Trends in parallelism and data

Number of Facebook Users

- 50 million
- 500 million

Average number of cores on TOP500

Jun-05 | Jun-06 | Jun-07 | Jun-08 | Jun-09 | Jun-10 | Jun-11
0       | 16     | 32     | 64     | 128    | 256    | 512

More cores and data \(\rightarrow\) Need to extract *algorithmic* parallelism
Graph abstractions in Computer Science

**Compiler optimization:**
Control flow graph: graph dominators
Register allocation: graph coloring

**Scientific computing:**
Preconditioning: support graphs, spanning trees
Sparse direct solvers: chordal graphs
Parallel computing: graph separators

**Computer Networks:**
Routing: shortest path algorithms
Web crawling: graph traversal
Interconnect design: Cayley graphs
Large graphs are everywhere

Internet structure
Social interactions

Scientific datasets: biological, chemical, cosmological, ecological, …

WWW snapshot, courtesy Y. Hyun

Yeast protein interaction network, courtesy H. Jeong
Types of graph computations

Examples:
- Centrality
- Shortest paths
- Network flows
- Strongly Connected Components

Tool: Graph Traversal

Examples:
- Loop and multi edge removal
- Triangle/Rectangle enumeration

Fuzzy intersection
Examples: Clustering, Algebraic Multigrid

Tightly coupled

Filtering based

Tools: Map/Reduce, SPARQL engines

Example:
- Tightly coupled

Examples:
- Filtering based
Types of graph computations

- Centrality
- Shortest paths
- Network flows
- Strongly Connected Components

Examples:

- Loop and multi edge removal
- Triangle/Rectangle enumeration

Challenges:
- Difficult to parallelize
- Very low arithmetic intensity
- Unpredictable data access patterns
“...my main conclusion after spending ten years of my life on the TeX project is that software is hard. It's harder than anything else I've ever had to do”
“...my main conclusion after spending ten years of my life on the TeX project is that software is hard. It's harder than anything else I've ever had to do”

Dealing with software is hard!
“...my main conclusion after spending ten years of my life on the TeX project is that software is hard. It's harder than anything else I've ever had to do”

Dealing with software is hard!

High performance computing (HPC) software is harder!
“...my main conclusion after spending ten years of my life on the TeX project is that software is hard. It's harder than anything else I've ever had to do”
Outline

- **Sustainable graph libraries**
- Linear algebraic primitives for graphs
- Sparse triple product for graph subroutines
- Filtered semantic graphs
- Future challenges/directions
Parallel Graph Analysis Software

Discrete structure analysis

Graph theory

Computers
Parallel Graph Analysis Software

Knowledge Discovery Toolbox (KDT)

Distributed Combinatorial BLAS

Shared-address space Combinatorial BLAS

Communication Support (MPI, GASNet, etc)

Threading Support (OpenMP, Cilk, etc)

Domain scientists

Graph algorithm developers

HPC scientists and engineers

Discrete structure analysis

Graph theory

Computers
Parallel Graph Analysis Software

- Knowledge Discovery Toolbox (KDT)
- Distributed Combinatorial BLAS
- Shared-address space Combinatorial BLAS
- Communication Support (MPI, GASNet, etc)
- Threading Support (OpenMP, Cilk, etc)

- Domain scientists
- Graph algorithm developers
- HPC scientists and engineers

- Graph theory
- Discrete structure analysis
- Computers

- KDT is higher level (graph abstractions)
- Combinatorial BLAS is for performance
(Semantic) directed graphs
- constructors, I/O
- basic graph metrics (e.g., \texttt{degree()})
- vectors

Clustering / components

Centrality / authority:
betweenness centrality, PageRank

Hypergraphs and sparse matrices

Graph primitives (e.g., \texttt{bfsTree()})

\texttt{SpMV} / \texttt{SpGEMM} on semirings
Domain expert vs. Graph expert

- **(Semantic) directed graphs**
  - constructors, I/O
  - basic graph metrics (*e.g.*, degree)
  - vectors
- **Clustering / components**
- **Centrality / authority:**
  betweenness centrality, PageRank
- **Hypergraphs and sparse matrices**
- **Graph primitives (*e.g.*, `bfsTree()`)**
- **SpMV / SpGEMM on semirings**

```python
# bigG contains the input graph
comp = bigG.connComp()
giantComp = comp.hist().argmax()
G = bigG.subgraph(comp==giantComp)

clus = G.cluster('Markov')
clusNedge = G.nedge(clus)
smallG = G.contract(clus)

# visualize
```
Domain expert vs. Graph expert

- (Semantic) directed graphs
  - constructors, I/O
  - basic graph metrics (e.g., degree, vectors)
- Clustering / components
- Centrality / authority: betweenness centrality, PageRank
- Hypergraphs and sparse matrices
- Graph primitives (e.g., bfsTree)
- SpMV / SpGEMM on semirings

```python
# bigG contains the input graph
comp = bigG.connComp()
giantComp = comp.hist().argmax()
G = bigG.subgraph(comp==giantComp)
clus = G.cluster('Markov')
clusNedge = G.nedge(clus)
smallG = G.contract(clus)

# visualize
```

```python
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)
```
Outline

- Sustainable graph libraries
- **Linear algebraic primitives for graphs**
- Sparse triple product for graph subroutines
- Filtered semantic graphs
- Future challenges/directions
The Combinatorial BLAS implements these, and more, on arbitrary semirings, e.g. \((\times, +), (\text{and}, \text{or}), (+, \text{min})\)
Many irregular applications contain coarse-grained parallelism that can be exploited by abstractions at the proper level.

<table>
<thead>
<tr>
<th>Traditional graph computations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data driven, unpredictable communication.</td>
</tr>
<tr>
<td>Irregular and unstructured, poor locality of reference</td>
</tr>
<tr>
<td>Fine grained data accesses, dominated by latency</td>
</tr>
</tbody>
</table>
Many irregular applications contain coarse-grained parallelism that can be exploited by abstractions at the proper level.

<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>
Breadth-first search in Combinatorial BLAS
Particular semiring operations:
**Multiply:** select
**Add:** minimum

parents:

from

\[
AT \\ X \quad \rightarrow
\]

to

\[
\begin{bmatrix}
A^T & X & A^T X
\end{bmatrix}
\]
Select vertex with minimum label as parent

parents:

from

to

\[ A^T \]
parents:
from 1 to 7

\[ \mathbf{A}^T \mathbf{X} \]
Matrix/vector distributions, interleaved on each other.

Default distribution in **Combinatorial BLAS**.

Scalable with increasing number of processes

- 2D matrix layout wins over 1D with large core counts and with limited bandwidth/compute
- 2D vector layout sometimes important for load balance

ALGORITHM:
1. Gather vertices in *processor column* [communication]
2. Find owners of the current frontier’s adjacency [computation]
3. Exchange adjacencies in *processor row* [communication]
4. Update distances/parents for unvisited vertices. [computation]
BFS strong scaling

- NERSC Hopper (Cray XE6, Gemini interconnect AMD Magny-Cours)
- Hybrid: In-node 6-way OpenMP multithreading
- Graph500 (R-MAT): 4 billion vertices and 64 billion edges.

Outline

- Sustainable graph libraries
- Linear algebraic primitives for graphs
- **Sparse triple product for graph subroutines**
- Filtered semantic graphs
- Future challenges/directions
Graph contraction via sparse triple product
Subgraph extraction via sparse triple product
Sparse matrix indexing (SpRef)

SpRef using mixed-mode sparse matrix-matrix multiplication (SpGEMM). Ex: \( B = A([2,4], [1,2,3]) \)

\[ T_{comm} = \Theta \left( \alpha \cdot \sqrt{p} + \beta \cdot \frac{nnz(A)}{\sqrt{p}} \right) \]

Algorithmic structure makes general case analysis possible

Dominated by SpGEMM

Bottleneck: bandwidth costs

Speedup: \( \Theta(\sqrt{p}) \)

Strong scaling of vertex relabeling

random symmetric permutation ⇔ relabeling graph vertices

- RMAT Scale 22; edge factor=8; a=.6, b=c=d=.4/3
- Franklin/NERSC, each node is a quad-core AMD Budapest
Parallel sparse matrix-matrix multiplication algorithms

\[ C_{ij} += \text{HyperSparseGEMM}(A^{\text{recv}}, B^{\text{recv}}) \]

2D algorithm: Sparse SUMMA (based on dense SUMMA)
General implementation that handles rectangular matrices

B., Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In ICPP’08
Local submatrix storage

Submatrices are “hypersparse” (i.e. \( \text{nnz} \ll n \))

\[ \sqrt{p} \] blocks

\[ \sqrt{p} \] blocks
Local submatrix storage

Submatrices are “hypersparse” (i.e. \( \text{nnz} \ll n \))

\[ \text{nnz}' = \frac{c}{\sqrt{p}} \to 0 \]

Average of \( c \) nonzeros per column
Submatrices are “hypersparse” (i.e. $nnz << n$)

Total Storage:

$O(n + nnz) \rightarrow O(n\sqrt{p} + nnz)$
Local submatrix storage

Submatrices are “hypersparse” (i.e. nnz << n)

\[ \frac{c}{\sqrt{p}} \rightarrow 0 \]

Average of c nonzeros per column

Total Storage:

\[ O(n + nnz) \rightarrow O(n\sqrt{p} + nnz) \]

- A data structure or algorithm that depends on matrix dimension n (e.g. CSR or CSC) is asymptotically too wasteful for submatrices
- Use doubly-compressed (DCSC) data structures or compressed sparse blocks (CSB) instead
Sequential “hypersparse” kernel

Operates on the strictly $O(nnz)$ DCSC data structure
Sparse outer-product formulation with multi-way merging
Efficient in parallel, i.e. $T(1) \approx p T(p)$

Time complexity:
$O(flops \cdot \lg n_i + nzc(A) + nzr(B))$
- independent of dimension

Space complexity:
$O(nnz(A) + nnz(B) + nnz(C))$
- independent of flops

1D vs. 2D scaling for sparse matrix-matrix multiplication

(a) R-MAT $\times$ R-MAT product (scale 21).  
(b) Multiplication of an R-MAT matrix of scale 23 with the restriction operator of order 8.

SpSUMMA = 2-D data layout (Combinatorial BLAS)  
EpetraExt = 1-D data layout (Trilinios)

In practice, 2D algorithms have the potential to scale, but not linearly

$$T_{\text{comm}}(2D) = \alpha p\sqrt{p} + \beta cn\sqrt{p}$$

$$T_{\text{comp}}(\text{optimal}) = c^2 n$$
Square sparse matrix multiplication

Almost linear scaling until bandwidth costs starts to dominate

Scaling proportional to $\sqrt{p}$ afterwards

NERSC/Franklin
Cray XT4

R-MAT, edgefactor: 8
$a=0.6$, $b=c=d=0.4/3$
Almost linear scaling until bandwidth costs starts to dominate

Scaling proportional to $\sqrt{p}$ afterwards

$$T_{\text{comm}} = \alpha p \sqrt{p} + \beta cn \sqrt{p}$$

$$T_{\text{comp}}(\text{optimal}) = c^2 n$$
Outline

• Sustainable graph libraries
• Linear algebraic primitives for graphs
• Sparse triple product for graph subroutines
• **Filtered semantic graphs**
• Future challenges/directions
Knowledge Discovery Toolbox
http://kdt.sourceforge.net/

A general graph library with operations based on linear algebraic primitives
Knowledge Discovery Toolbox
http://kdt.sourceforge.net/

- Aimed at domain experts who know their problem well but don’t know how to program a supercomputer
- Easy-to-use Python interface
- Runs on a laptop as well as a cluster with 10,000 processors

A general graph library with operations based on linear algebraic primitives
Knowledge Discovery Toolbox
http://kdt.sourceforge.net/

- Aimed at domain experts who know their problem well but don’t know how to program a supercomputer
- Easy-to-use Python interface
- Runs on a laptop as well as a cluster with 10,000 processors

- A collaboration among LBNL, UCSB, UC Berkeley, and Cray Inc.
- Open source software, released under New BSD license
- v0.1 released March 2011; v0.2 released March 2012

Lugowski, Alber, B., Gilbert, Reinhardnt, Teng, and Waranis. A flexible open-source toolbox for scalable complex graph analysis. *SIAM Conference on Data Mining (SDM), 2012*
How to target “domain experts”? 

- Performance  
- Conceptual simplicity  
- Customizability  

Combinatorial BLAS

KDT with just in time compilation

KDT’s non-semantic graphs

KDT’s semantic graphs
The need for filters

Graph of text & phone calls

Betweenness centrality

Betweenness centrality on text messages

Betweenness centrality on phone calls
Example:

- Vertex types: Person, Phone, Camera, Gene, Pathway
- Edge types: PhoneCall, TextMessage, CoLocation, Sequence Similarity
- Edge attributes: StartTime, EndTime
- Calculate centrality just for emails among engineers sent between times sTime and eTime

```python
def onlyEngineers(self):
    return self.position == Engineer

def timedEmail(self, sTime, eTime):
    return ((self.type == email) and (self.Time > sTime) and (self.Time < eTime))

start = dt.now() - dt.timedelta(days=30)
end = dt.now()

# G denotes the graph
G.addVFilter(onlyEngineers)
G.addEFilter(timedEmail(start, end))

# rank via centrality based on recent email transactions among engineers
bc = G.rank('approxBC')
```

- Lugowski, B., Gilbert, Reinhardt. Scalable complex graph analysis with the knowledge discovery toolbox. In ICASSP, 2012
Edge filter illustration

class edge_attr:
  isText
  isPhoneCall
  weight
G.addEFilter(lambda e: e.weight > 0)

class edge_attr:
    isText
    isPhoneCall
    weight
class edge_attr:
    isText
    isPhoneCall
    weight

G.addEFilter(lambda e: e.weight > 0)
G.addEFilter(lambda e: e.isPhoneCall)
Filter options and implementation

- Filter defined as unary predicates, checked in order they were added
- **Each KDT object** maintains a stack of **filter predicates**
- All operations respect filters, enabling **filter-ignorant algorithm design**

<table>
<thead>
<tr>
<th>On-the-fly filters</th>
<th>Materialized filters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edges are retained</td>
<td>Edges are pruned on copy</td>
</tr>
<tr>
<td>Check predicate on each edge/vertex traversal</td>
<td>Check predicate once on materialization</td>
</tr>
<tr>
<td>Cheap but done on each run</td>
<td>Expensive but done once</td>
</tr>
</tbody>
</table>
Selective Embedded Just-In-Time Specialization

- Take Python code
- Translate it to equivalent C++ code
- Compile with GCC
- Call compiled version instead of Python version

On-the-fly filters need SEJITS for fast execution:

- In plain KDT, filters are pure Python functions.
- Requires a per-vertex or per-edge upcall into Python
- Can be as slow as 80X compared to pure C++

- B., Gilbert, Fox, Kamil, Lugowski, Oliker, Williams. High-Performance Analysis of Filtered Semantic Graphs, *Tech Report UCB/EECS-2012-61 (Extended Abstract at PACT’12)*
Filtered BFS with SEJITS

Time (in seconds) for a single BFS iteration on Scale 25 RMAT (33M vertices, 500M edges) with 10% of elements passing filter. Machine is NERSC’s Hopper.
Filtered BFS with SEJITS

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.
Outline

• Sustainable graph libraries
• Linear algebraic primitives for graphs
• Sparse triple product for graph subroutines
• Filtered semantic graphs
• Future challenges/directions
Challenges #1: Communication

Communication is the bottleneck!

2.5D Algorithms:
Using extra memory (c replicas) reduces communication volume by a factor of $c^{1/2}$ compared to 2D

Scale 23 R-MAT times restriction operator of order 4
Challenges #2: Data diversity

Low diameter graph (R-MAT) vs. Long skinny graph (genomics)

Gene linkage map, courtesy Yan et al.
Challenges #3: End to end solution

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

The whole pipeline should be integrated with KDT
Challenges #4: Attack the yet undefined

- High-performance graph analysis should help define yet “undefined” data analysis problems in sciences.
- Key: Collaborating directly with domain scientists.

Example: Pan-genome analysis

Microbial pan-genome, courtesy K. Ning
Conclusions

- **KDT + Combinatorial BLAS**: Making parallel graph analysis accessible to domain scientists.
- Layered **software architecture** allows concurrent advances in performance and functionality.
- Carefully chosen and optimized **primitives** are the key to high performance and scalability in graph computations.
- **Linear algebraic primitives** of the Combinatorial BLAS are general enough to be widely useful and compact enough to be heavily optimized.
- High-performance **filtered semantic graph processing** is possible without changes from the graph algorithm developer.
Acknowledgments...

Combinatorial BLAS class hierarchy

Combinatorial BLAS functions and operators

- DistMat
- CommGrid
- FullyDistVec

- DenseDistMat
- SpDistMat
- SpMat
- SpDistVec
- DenseDistVec

- DCSC
- CSC
- Triples
- CSB

Enforces interface only

Polymorphism

HAS A
Some Combinatorial BLAS functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Applies to</th>
<th>Parameters</th>
<th>Returns</th>
<th>Matlab Phrasing</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpGEMM</td>
<td>Sparse Matrix (as friend)</td>
<td>A, B: Sparse matrices, trA: transpose A if true, trB: transpose B if true</td>
<td>Sparse Matrix</td>
<td>C = A * B</td>
</tr>
<tr>
<td>SpMV</td>
<td>Sparse Matrix (as friend)</td>
<td>A: Sparse matrices, x: sparse or dense vector(s), trA: transpose A if true</td>
<td>Sparse or Dense</td>
<td>y = A * x</td>
</tr>
<tr>
<td>SpEWiseX</td>
<td>Sparse Matrices (as friend)</td>
<td>A, B: Sparse matrices, notA: negate A if true, notB: negate B if true</td>
<td>Sparse Matrix</td>
<td>C = A * B</td>
</tr>
<tr>
<td>REDUCE</td>
<td>Any Matrix (as method)</td>
<td>dim: dimension to reduce</td>
<td>Dense Vector</td>
<td>sum(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>binop: reduction operator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpRef</td>
<td>Sparse Matrix (as method)</td>
<td>p: row indices vector, q: column indices vector</td>
<td>Sparse Matrix</td>
<td>B = A(p, q)</td>
</tr>
<tr>
<td>SpAsgn</td>
<td>Sparse Matrix (as method)</td>
<td>p: row indices vector, q: column indices vector, B: matrix to assign</td>
<td>none</td>
<td>A(p, q) = B</td>
</tr>
<tr>
<td>Scale</td>
<td>Any Matrix (as method)</td>
<td>rhs: any object (except a sparse matrix)</td>
<td>none</td>
<td>Check guiding principles 3 and 4</td>
</tr>
<tr>
<td>Scale</td>
<td>Any Vector (as method)</td>
<td>rhs: any vector</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>APPLY</td>
<td>Any Object (as method)</td>
<td>unop: unary operator (applied to non-zeros)</td>
<td>None</td>
<td>none</td>
</tr>
</tbody>
</table>
Recursive all-pairs shortest paths

\[ A = A*; \quad \% \text{recursive call} \]
\[ B = AB; \quad C = CA; \]
\[ D = D + CB; \]
\[ D = D*; \quad \% \text{recursive call} \]
\[ B = BD; \quad C = DC; \]
\[ A = A + BC; \]

+ is “min”, × is “add”
Novel 2.5D APSP algorithm
[Solomonik, B., Demmel; 2012]

Bandwidth: \( W_{bc-2.5D}(n, p) = O(n^2 / \sqrt{cp}) \)

Latency: \( S_{bc-2.5D}(p) = O(\sqrt{cp \log^2(p)}) \)

Optimal for any memory size!
Novel 2.5D APSP algorithm

Number of compute nodes

GFlops

n=4096

n=8192
### Complex methods

- centrality('exactBC')
- centrality('approxBC')
- cluster('Markov')
- pageRank

### Building blocks

<table>
<thead>
<tr>
<th>DiGraph</th>
<th>HyGraph</th>
<th>SpMatrix</th>
<th>SpVec</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfsTree, neighbor</td>
<td>toDiGraph</td>
<td>SpMV</td>
<td>max, norm, sort</td>
</tr>
<tr>
<td>degree, subgraph</td>
<td>load, Ufget</td>
<td>SpGEMM</td>
<td>abs, any, ceil</td>
</tr>
<tr>
<td>load, UFget</td>
<td>bfsTree</td>
<td>load, eye</td>
<td>range, ones</td>
</tr>
<tr>
<td>+, -, sum, scale</td>
<td>degree</td>
<td>reduce, scale</td>
<td>+, -, *, /, &gt;, ==, &amp;</td>
</tr>
</tbody>
</table>

### Underlying infrastructure (Combinatorial BLAS)

- SpMV, SpMV_SemiRing
- SpGEMM, SpGEMM_SemiRing
- Sparse-matrix classes/methods (e.g., Apply, EWiseApply, Reduce)