Three Goals in Parallel Graph Computations: High Performance, High Productivity, and Reduced Communication

Aydın Buluç
Berkeley Lab (LBNL)
October 21, 2013
Acknowledgments (past & present)

- Krste Asanovic (UC Berkeley)
- Grey Ballard (UC Berkeley)
- Scott Beamer (UC Berkeley)
- Jim Demmel (UC Berkeley)
- Erika Duriakova (UC Dublin)
- Armando Fox (UC Berkeley)
- John Gilbert (UCSB)
- Laura Grigori (INRIA)
- Shoaib Kamil (MIT)
- Ben Lipshitz (UC Berkeley)
- Adam Lugowski (UCSB)
- Kamesh Madduri (Penn State)
- Lenny Oliker (Berkeley Lab)
- Dave Patterson (UC Berkeley)
- Steve Reinhardt (YarcData)
- Oded Schwartz (UC Berkeley)
- Edgar Solomonik (UC Berkeley)
- Veronika Strnadova (UCSB)
- Sivan Toledo (Tel Aviv Univ)
- Sam Williams (Berkeley Lab)

This work is funded by:

U.S. Department of Energy
Office of Science
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

Tools: Graph Traversal

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

Tightly coupled

Filtering based

Fuzzy intersection

Examples: Clustering, Algebraic Multigrid

Tools: Map/Reduce, SPARQL engines

Examples:
- Loop and multi edge removal
- Triangle/Rectangle enumeration
Types of graph computations

Examples:
- 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

Tool: Graph Traversal

Tightly coupled

Fuzzy intersection

Examples: Clustering, Algebraic Multigrid
Part 1: High performance

Crux: Linear-algebraic primitives
The Combinatorial BLAS implements these, and more, on arbitrary semirings, e.g. \((\times, +), (\text{and}, \text{or}), (+, \text{min})\)
**Some Combinatorial BLAS functions**

* Grossly simplified (parameters, semiring)

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Returns</th>
<th>Math Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpGEMM</td>
<td>- sparse matrices A and B</td>
<td>sparse matrix</td>
<td>C = op(A) * op(B)</td>
</tr>
<tr>
<td></td>
<td>- unary functors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpM{S/D}V</td>
<td>- sparse matrix A</td>
<td>sparse/dense vector</td>
<td>y = A * x</td>
</tr>
<tr>
<td></td>
<td>- sparse/dense vector x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpEWiseX</td>
<td>- sparse matrices or vectors</td>
<td>in place or sparse matrix/vector</td>
<td>C = A .* B</td>
</tr>
<tr>
<td></td>
<td>- binary functor and predicate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reduce</td>
<td>- sparse matrix A and functors</td>
<td>dense vector</td>
<td>y = sum(A, op)</td>
</tr>
<tr>
<td>SpRef</td>
<td>- sparse matrix A</td>
<td>sparse matrix</td>
<td>B = A(p,q)</td>
</tr>
<tr>
<td></td>
<td>- index vectors p and q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpAsgn</td>
<td>- sparse matrices A and B</td>
<td>none</td>
<td>A(p,q) = B</td>
</tr>
<tr>
<td></td>
<td>- index vectors p and q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scale</td>
<td>- sparse matrix A</td>
<td>none</td>
<td>check manual</td>
</tr>
<tr>
<td></td>
<td>- dense matrix or vector X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apply</td>
<td>- any matrix or vector X</td>
<td>none</td>
<td>op(X)</td>
</tr>
</tbody>
</table>
1. \( S = \{ \} ; \ C = \{ 1, 2, 3, 4, 5, 6, 7, 8 \} \);

2. while \( C \) is not empty {

3. label each \( v \) in \( C \) with a random \( r(v) \);

4. for all \( v \) in \( C \) in parallel {

5. if \( r(v) < \min(\ r(\text{neighbors of } v) \) ) {

6. move \( v \) from \( C \) to \( S \);

7. remove neighbors of \( v \) from \( C \);

8. }

9. }

10. }

1. $S = \text{empty set}; \ C = V;$

2. while $C$ is not empty {

3. label each $v$ in $C$ with a random $r(v)$;

4. for all $v$ in $C$ in parallel {

5. if $r(v) < \min( r(\text{neighbors of } v) )$ {

6. move $v$ from $C$ to $S$;

7. remove neighbors of $v$ from $C$;

8. }

9. }

10. }

$S = \{ 1, 5 \}$

$C = \{ 6, 8 \}$
1. \( S = \) empty set; \( C = V \);
2. while \( C \) is not empty {
3.     label each \( v \) in \( C \) with a random \( r(v) \);
4.     for all \( v \) in \( C \) in parallel {
5.         if \( r(v) < \min( r(\text{neighbors of } v) ) \) {
6.             move \( v \) from \( C \) to \( S \);
7.             remove neighbors of \( v \) from \( C \);
8.         }
9.     }
10. }

\[ S = \{ 1, 5 \} \]
\[ C = \{ 6, 8 \} \]
Parallel, randomized maximal independent set algorithm

1. \( S = \text{empty set}; \ C = V; \)
2. while \( C \) is not empty {
3. \hspace{1em} \text{label each } v \text{ in } C \text{ with a random } r(v); \\
4. \hspace{1em} \text{for all } v \text{ in } C \text{ in parallel} \
5. \hspace{2em} \text{if } r(v) < \min( r(\text{neighbors of } v) ) \} \\6. \hspace{2em} \text{move } v \text{ from } C \text{ to } S; \\
7. \hspace{1em} \text{remove neighbors of } v \text{ from } C; \\
8. \hspace{1em} }
9. }
10. }

Theorem: This algorithm “very probably” finishes within \( O(\log n) \) rounds.

\[
S = \{ 1, 5, 8 \} \\
C = \{ \}
\]
Luby’s algorithm in KDT

Input: $A^T$ (transpose of Graph’s adjacency matrix)
Output: MIS (maximal independent set)

while (candidates.nnz() > 0): # number of nonzeros
    candidates.Apply (rand) # label each vertex in candidates with random value
    # find the smallest random value among a vertex's neighbors
    minadj = $A^T$.SpSMV (candidates, semiring(min,select2nd))

    # Add vertices to S if its -own value-- is smallest among neighbors
    newS = minadj.SpEWiseX (candidates, op=[{}{return 1}, predicate=is2ndSmaller,...)

    # newS are no longer candidates, so remove them
    candidates. SpEWiseX (newS, op=[{}{return 1}, ...) # in place

    sadj = $A^T$.SpSMV (newS, semiring(select2nd,select2nd)) # find neighbors of newS

    # remove sadj from candidate as well; they can’t be part of the MIS
    candidates. SpEWiseX (sadj, op=[{}{return 1}, ...) # in place

    MIS += newS # add new_S to MIS
Graph algorithm comparison (LA: linear algebra)

<table>
<thead>
<tr>
<th>Algorithm (Problem)</th>
<th>Canonical Complexity</th>
<th>LA-Based Complexity</th>
<th>Critical Path (for LA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-first search</td>
<td>$\Theta(m)$</td>
<td>$\Theta(m)$</td>
<td>$\Theta(\text{diameter})$</td>
</tr>
<tr>
<td>Betweenness Centrality (unweighted)</td>
<td>$\Theta(mn)$</td>
<td>$\Theta(mn)$</td>
<td>$\Theta(\text{diameter})$</td>
</tr>
<tr>
<td>All-pairs shortest-paths (dense)</td>
<td>$\Theta(n^3)$</td>
<td>$\Theta(n^3)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Prim (MST)</td>
<td>$\Theta(m+n \log n)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Borůvka (MST)</td>
<td>$\Theta(m \log n)$</td>
<td>$\Theta(m \log n)$</td>
<td>$\Theta(\log^2 n)$</td>
</tr>
<tr>
<td>Edmonds-Karp (Max Flow)</td>
<td>$\Theta(m^2n)$</td>
<td>$\Theta(m^2n)$</td>
<td>$\Theta(mn)$</td>
</tr>
<tr>
<td>Greedy MIS (MIS)</td>
<td>$\Theta(m+n \log n)$</td>
<td>$\Theta(mn+n^2)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Luby (MIS)</td>
<td>$\Theta(m+n \log n)$</td>
<td>$\Theta(m \log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
</tbody>
</table>

Majority of selected algorithms can be represented with array-based constructs with equivalent complexity. ($n = |V|$ and $m = |E|$)
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>
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]
Part 2: High productivity

Crux: Very high-level languages and selective just-in-time translation using domain-specific languages.
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

Parallel computing experts

Discrete structure analysis

Graph theory

Computers

- KDT is higher level (graph abstractions)
- Combinatorial BLAS is for performance
How to target “domain experts”? 

Performance  
Conceptual simplicity 
Customizability

Combinatorial BLAS

KDT’s non-semantic graphs
KDT with just in time compilation
KDT’s semantic graphs

KDT's non-semantic graphs
Attributed semantic graphs and filters

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')
```

Edge filter illustration

class edge_attr:
    isText
    isPhoneCall
    weight
Edge filter illustration

G.addEFilter(lambda e: e.weight > 0)

class edge_attr:
  isText
  isPhoneCall
  weight
G.addEFilter(lambda e: e.weight > 0)
G.addEFilter(lambda e: e.isPhoneCall)
Problems with Customizing in KDT

• Filtering on attributed semantic graphs is slow
  • 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++

• Adding new graph algorithms to KDT is slow
  • A new graph algorithm = composing linear algebraic primitives + customizing the semiring operation
  • Semirings in Python; similar performance bottleneck
Review: Selective Embedded Just In Time Specialization (SEJITS)

SEJITS for filter/semiring acceleration

Standard KDT

Python

KDT Algorithm

Filter (Py)

Semiring (Py)

C++

ComblLAS Primitive
SEJITS for filter/semiring acceleration

**Standard KDT**
- KDT Algorithm
- Filter (Py)
- Semiring (Py)

**KDT+SEJITS**
- KDT Algorithm
- Filter (Py)
- Semiring (Py)

**SEJITS Translation**
- Filter (C++)
- Semiring (C++)

**Embedded DSL: Python for the whole application**
- Introspect, translate Python to equivalent C++ code
- Call compiled/optimized C++ instead of Python

B., Duriakova, Gilbert, Fox, Kamil, Lugowski, Oliker, Williams. High-Performance and High-Productivity Analysis of Filtered Semantic Graphs, *IPDPS*, 2013
**SEJITS+KDT multicore performance**

![Graph showing the performance of SEJITS+KDT with different filter permeabilities](image)

- MIS= Maximal Independent Set
- 36 cores of Mirasol (Intel Xeon E7-8870)
- Erdős-Rényi (Scale 22, edgefactor=4)

**Synthetic data with weighted randomness to match filter permeability**

Notation: [semiring impl] / [filter impl]
A **roofline model** for shows how SEJITS moves KDT analytics from being Python **compute bound** to being **bandwidth bound**.

- Breadth-first search
- 576 cores of Hopper (Cray XE6 at NERSC with AMD Opterons)
- R-MAT (Scale 25, edgefactor=16, symmetric)
Part 3: Minimal Communication

Crux: Communication-avoiding versions of higher-level level primitives
Two kinds of costs:
- Arithmetic (FLOPs)
- Communication: moving data

Running time $= \gamma \cdot \#FLOPs + \beta \cdot \#Words + (\alpha \cdot \#Messages)$

**Sequential**
- CPU
- RAM
- 23%/Year
- 59%/Year

**Distributed**
- $P$ processors
- 26%/Year
- 59%/Year

2004: trend transition into multi-core, further communication costs

Develop faster algorithms: minimize communication (to lower bound if possible)
- Often no surface to volume ratio.
- Very little data reuse in existing algorithmic formulations *
- Already heavily communication bound

2D sparse matrix-matrix multiply emulating:
- Graph contraction
- AMG restriction operations

Scale 23 R-MAT (scale-free graph) **times** order 4 restriction operator

Cray XT4, Franklin, NERSC
All-pairs shortest-paths problem

- **Input:** Directed graph with “costs” on edges
- **Find least-cost paths between all reachable vertex pairs**
- **Classical algorithm:** Floyd-Warshall

```plaintext
for k=1:n  // the induction sequence
  for i = 1:n
    for j = 1:n
      if(w(i→k)+w(k→j) < w(i→j))
        w(i→j):= w(i→k) + w(k→j)
```

- **k = 1 case**

- It turns out a previously overlooked **recursive version** is more parallelizable than the triple nested loop
\[
\begin{bmatrix}
0 & 5 & 9 & \infty & \infty & 4 \\
\infty & 0 & 1 & -2 & \infty & \infty \\
3 & \infty & 0 & 4 & \infty & 3 \\
\infty & \infty & 5 & 0 & 4 & \infty \\
\infty & \infty & -1 & \infty & 0 & \infty \\
-3 & \infty & \infty & \infty & 7 & 0 \\
\end{bmatrix}
\]

\( + \) is "min", \( \times \) is "add"

\[
A = A*; \quad \% \text{recursive call}
\]
\[
B = AB; \quad C = CA;
\]
\[
D = D + CB;
\]
\[
D = D*; \quad \% \text{recursive call}
\]
\[
B = BD; \quad C = DC;
\]
\[
A = A + BC;
\]
The cost of 3-1-2 path

\[ a(3, 2) = a(3, 1) + a(1, 2) \quad \text{then} \quad \Pi(3, 2) = \Pi(1, 2) \]
D = D*: no change

\[
\begin{bmatrix}
5 & 9 \\
8 & 0
\end{bmatrix}
\begin{bmatrix}
0 & 1 \\
8 & 0
\end{bmatrix}
= \begin{bmatrix}
5 & 6
\end{bmatrix}
\]

B

D

Path: 1-2-3

\[a(1,3) = a(1,2) + a(2,3)\]

then \[\Pi(1,3) = \Pi(2,3)\]
Communication-avoiding APSP on distributed memory

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

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

c: number of replicas

Optimal for any memory size!
Communication-avoiding APSP on distributed memory

Questions?
Graph contraction via sparse triple product

Contract

A1
A2
A3

A1
A2
A3

1 2 3 4 5 6
1 1 1 1 1 1
2 1 1
3 1 1

1 2 3 4 5 6
1
2
3
4
5
6

1 1 1
1 1