Scaling Betweenness Centrality using Communication-Efficient Sparse Matrix Multiplication

Edgar Solomonik[1,2], Maciej Besta[1], Flavio Vella[1], and Torsten Hoefler[1]

[1] Department of Computer Science
ETH Zurich

[2]Department of Computer Science
University of Illinois at Urbana-Champaign

Outline

1. Betweenness Centrality
   - Problem Definition
   - All-Pairs Shortest-Paths
   - Brandes' Algorithm
   - Parallel Brandes' Algorithm
2. Sparse Matrix Multiplication
   - Algebraic Shortest Path Computation
   - Parallel Sparse Matrix Multiplication
3. Algebraic Parallel Programming
   - Cyclops Tensor Framework
   - Performance Results
4. Conclusion

Centrality in Graphs

Betweenness centrality – For each vertex v in $G = (V, E)$, sum the fractions of shortest paths $s \sim t$ that pass through v,

$$\lambda(v) = \sum_{s,t \in V} \sigma_v(s,t)/\sigma(s,t).$$

- $\sigma(s,t)$ is the number (**multiplicity**) of shortest paths $s \sim t$
- $\sigma_v(s,t)$ is the number of shortest paths $s \sim t$ that pass through v
- Shortest paths can be **unweighted** or **weighted**
- Centrality is important in analysis of biology, transport, and social network graphs

Path Multiplicities

- Let $d(s,t)$ be the shortest distance between vertex s and vertex t
- The multiplicity of shortest paths $\sigma(s,t)$ is the number of distinct paths $s \sim t$ with distance $d(s,t)$
- If v is in some shortest path $s \sim t$, then

$$d(s,t) = d(s,v) + d(v,t)$$

- Consequently, can compute all $\sigma_v(s,t)$ and $\lambda(v)$ given all distances

$$\sigma_v(s,t) = \begin{cases} \sigma(s,v)\sigma(v,t) & : d(s,t) = d(s,v) + d(v,t) \\ 0 & : \text{otherwise} \end{cases}$$

Betweenness Centrality by All-Pairs Shortest-Paths

- We can obtain $d(s,t)$ for all s,t by all-pairs shortest-paths (**APSP**)
- Multiplicities (σ and σ_v for each v) are easy to get given distances
- However, the cost of APSP is prohibitive, for n-node graphs:
    - $Q = \Theta(n^3)$ work with typical algorithms (e.g. Floyd-Warshall)
    - $D = \Theta(\log(n))$ depth[1]
    - $M = \Theta(n^2/p)$ memory footprint per processor
- **APSP does not effectively exploit graph sparsity**

[1] Tiskin, Alexander. "All-pairs shortest paths computation in the BSP model." Automata, Languages and Programming (2001): 178-189.

Brandes' Algorithm for Betweenness Centrality

Ulrik Brandes proposed a **memory-efficient** method[1]

- Compute $d(s,\star)$ and $\sigma(s,\star)$ for a given source vertex s
- Using these calculate **partial centrality factors** $\zeta(s,v)$ so

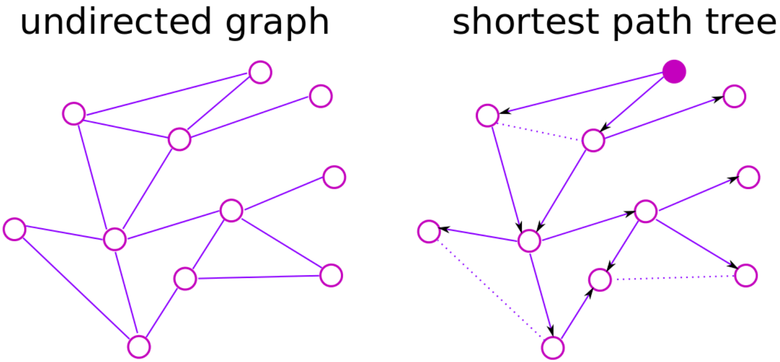
$$\zeta(s,v) = \sum_{t \in V, d(s,v)+d(v,t)=d(s,t)} \sigma(v,t)/\sigma(s,t)$$

- Construct the centrality scores from partial centrality factors

$$\lambda(v) = \sum_{s} \sigma(s,v)\zeta(s,v)$$

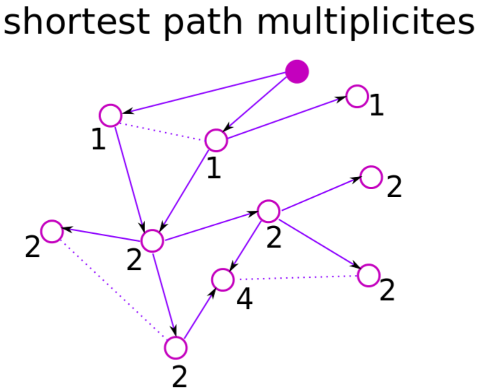
[1] Brandes, Ulrik. "A faster algorithm for betweenness centrality." Journal of mathematical sociology 25.2 (2001): 163-177.

Shortest Path Tree



If any multiplicity $\sigma(s,t) > 1$, shortest path tree has cross edges

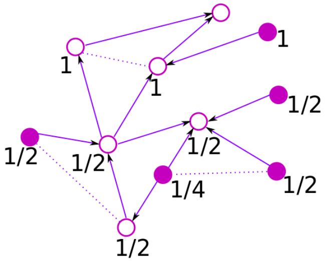
Shortest Path Tree Multiplicities



$\sigma(s, v)$ value displayed for each node v given colored source vertex s

Partial Centrality Factors in Shortest Path Tree

betweenness centrality back-propagation



If $\pi(s, v)$ are the children of v in shortest path tree from s

$$\zeta(s, v) = \sum_{c \in \pi(s,v)} \left(\frac{1}{\sigma(s, c)} + \zeta(s, c) \right)$$

Brandes' Algorithm Overview

- For each source vertex $s \in V$ (or a **batch** of source vertices)
- Compute single-source shortest-paths (**SSSP**) from s
  - For unweighted graphs, use breadth first search (**BFS**)
  - More viable choices for weighted graphs: Dijkstra, **Bellman-Ford**, Δ-stepping, ...
- Perform **back-propagation** of centrality scores on shortest path tree from s
  - Roughly as hard as BFS regardless of whether G is weighted

Parallelism in Brandes' Algorithm

Sources of parallelism in Brandes' algorithm:

- Computation of SSSP and back-propagation
  - Concurrency and efficiency like **BFS** on graphs
  - **Bellman-Ford provides maximal concurrency** for weighted graphs at cost of extra work
- Different source vertices can be processed in parallel as a batch
  - Key **additional source of concurrency**
  - Maintaining more distances requires **greater memory footprint**, $M = \Omega(bn/p)$ for batch size b

Algebraic shortest path computations

Tropical (geodetic) semiring

- additive (idempotent) operator: $a \oplus b = \min(a, b)$, identity: ∞
- multiplicative operator: $a \otimes b = a + b$, identity: 0
- matrix multiplication defined accordingly,

$$\boldsymbol{C} = \boldsymbol{A} \otimes \boldsymbol{B} \quad \Rightarrow \quad c_{ij} = \min_k (a_{ik} + b_{kj})$$

Algebraic shortest path computations

Tropical (geodetic) semiring

- additive (idempotent) operator: $a \oplus b = \min(a, b)$, identity: ∞
- multiplicative operator: $a \otimes b = a + b$, identity: 0
- matrix multiplication defined accordingly,

$$\boldsymbol{C} = \boldsymbol{A} \otimes \boldsymbol{B} \quad \Rightarrow \quad c_{ij} = \min_k (a_{ik} + b_{kj})$$

Bellman-Ford algorithm (SSSP) for $n \times n$ adjacency matrix $\boldsymbol{A}$:

1. initialize $\boldsymbol{v}^{(1)} = (0, \infty, \infty, \ldots)$
2. compute $\boldsymbol{v}^{(n)}$ via recurrence

$$\boldsymbol{v}^{(i+1)} = \boldsymbol{v}^{(i)} \oplus (\boldsymbol{A} \otimes \boldsymbol{v}^{(i)})$$

Algebraic View of Brandes' Algorithm

- Given frontier vector $\boldsymbol{x}^{(i)}$ and tentative distances $\boldsymbol{w}^{(i)}$

$$\boldsymbol{y}^{(i)} = \boldsymbol{A} \otimes \boldsymbol{x}^{(i)} \quad \text{and} \quad \boldsymbol{w}^{(i+1)} = \boldsymbol{w}^{(i)} \oplus \boldsymbol{y}^{(i)}$$

- $\boldsymbol{x}^{(i+1)}$ given by entries if $\boldsymbol{w}^{(i+1)}$ that differ from $\boldsymbol{w}^{(i)}$
- For BFS, each tentative distance changes only once
- For Bellman-Ford, tentative distances can change multiple times
- Thus both algorithms require iterative **SpMSpV**
- Having a batch size $b > 1$ transforms the problem to **sparse matrix multiplication (SpGEMM or SpMSpM)**

Communication Avoiding Sparse Matrix Multiplication

- Let the bandwidth cost W be the maximum amount of data communicated by any processor
- We use analogue of 1D/2D/3D rectangular matrix multiplication
- The bandwidth cost of matrix multiplication $\boldsymbol{Y} = \boldsymbol{A}\boldsymbol{X}$ is then

$$W = \min_{p_1p_2p_3=p} \left[\frac{\text{nnz}(\boldsymbol{A})}{p_1p_2} + \frac{\text{nnz}(\boldsymbol{X})}{p_2p_3} + \frac{\text{nnz}(\boldsymbol{Y})}{p_1p_3} \right] \Bigg)$$

- In our context, $\text{nnz}(\boldsymbol{A}) = |E| = m$, while $\boldsymbol{X}$ holds current frontiers for b starting vertices, so $\text{nnz}(\boldsymbol{X}) \leq nb$

Communication Avoiding Betweenness Centrality

- Latency cost is proportional to number of SpMSpM calls
- Replication of $\boldsymbol{A}$ for SpMSpMs minimizes bandwidth cost W
- It then suffices to communicate frontiers $\boldsymbol{X}$ and reduce results $\boldsymbol{Y}$
- For undirected graphs, for b starting vertices, **total nonzeros in $\boldsymbol{X}$ over all iterations** is nb and for $\boldsymbol{Y}$ is $O(nb)$
- Best choice of b with sufficient memory gives

$$W = O(n\sqrt{m}/p^{2/3})$$

- Memory-limited communication cost bound given in paper

Cyclops Tensor Framework (CTF) [1]

- Distributed-memory symmetric/sparse tensors in C++ or Python
- For betweenness centrality, we only use CTF matrices

```
Matrix<int> A(n, n, AS|SP, World(MPI_COMM_WORLD));
A.read(...); A.write(...); A.slice(...); A.permute(...);
```

- Matrix **summation** in CTF notation is

```
B["ij"] += A["ij"];
```

- Matrix **multiplication** in CTF notation is

```
Y["ij"] += T["ik"]*X["kj"];
```

- **Used-defined elementwise functions** can be used with either

```
Y["ij"] += Function<>([](double x){ return 1/x; })(X["ij"]);
Y["ij"] += Function<int,double,double>(...)(A["ik"],X["kj"]);
```

[1] E. Solomonik, D. Matthews, J. Hammond, J. Demmel, JPDC 2014

CTF Code for Betweenness Centrality

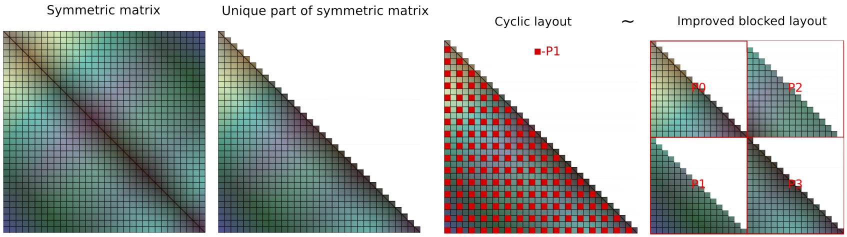
```
void btwn_central(Matrix<int> A, Matrix<path> P, int n){
  Monoid<path> mon(...,
                 [](path a, path b){
                   if (a.w<b.w) return a;
                   else if (b.w<a.w) return b;
                   else return path(a.w, a.m+b.m);
                 }, ...);

  Matrix<path> Q(n,k,mon); // shortest path matrix
  Q["ij"] = P["ij"];

  Function<int,path> append([](int w, path p){
                     return path(w+p.w, p.m);
                   }; );

  for (int i=0; i<n; i++)
    Q["ij"] = append(A["ik"],Q["kj"]);
  ...
}
```

Symmetry and Sparsity by Cyclicity



A **cyclic** layout provides

- preservation of packed symmetric storage format
- **load balance** for sparse 1D/2D (vertex/edge) graph blocking
- **obliviousness** with respect to **graph structure/topology**

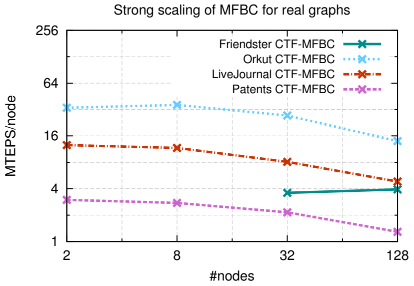
Data Mapping and Autotuning

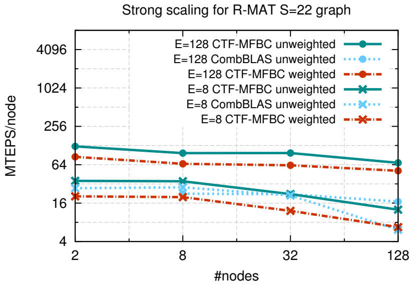
The CTF workflow is as follows

- All operations executed bulk synchronously
- For each product, matrices can be redistributed globally
- Arbitrary sparsity supported via compressed-sparse-row (CSR)
  - Modularity permits alternative sparse matrix representaitons
- Performance model used to select best contraction algorithm
  - Leverages randomized distribution of nonzeros (edges)
  - Model coefficients tuned using linear regression
- Layout and algorithm choices are made **at runtime** using model

CTF Performance for Betweenness Centrality

- Implementation uses CTF SpGEMM adaptively with **sparse or dense output (push or pull)**
- We compare with **CombBLAS**, which uses semirings and BFS (unweighted only)





Friendster has 66 million vertices and 1.8 billion edges (results on Blue Waters, Cray XE6)

Conclusions and Future Work

- Summary of algorithmic contributions
  - Parallel **communication-avoiding** betwenness centrality algorithm
  - **Better** sparse matrix multiplication for unbalanced nonzero counts
  - Algorithms and implementation general to **weighted** graphs
- Future work
  - Use of Δ-stepping or other more work-efficient SSSP algorithms
  - Optimizations in conjunction with approximation algorithms

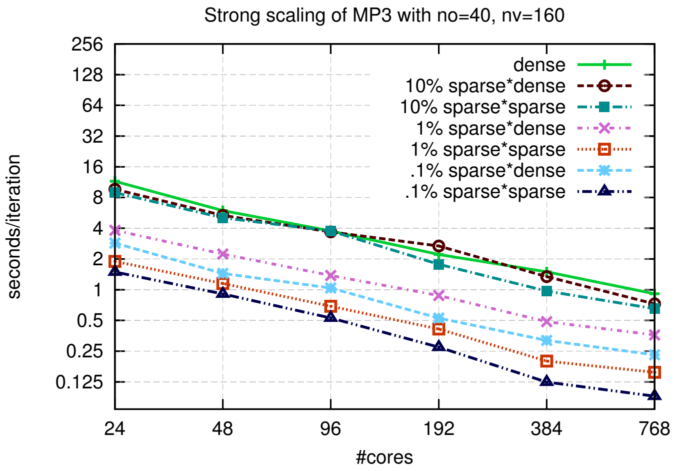
Cyclops Tensor Framework

- Graphs are **one of many applications**, other highlights include
  - **Petascale** high-accuracy quantum chemistry
  - **56-qubit** (largest ever) quantum computing simulation
- Already provides most functionality proposed in GraphBLAS 1, plus all of that for tensors (hypergraphs with uniform size nets)

Backup slides

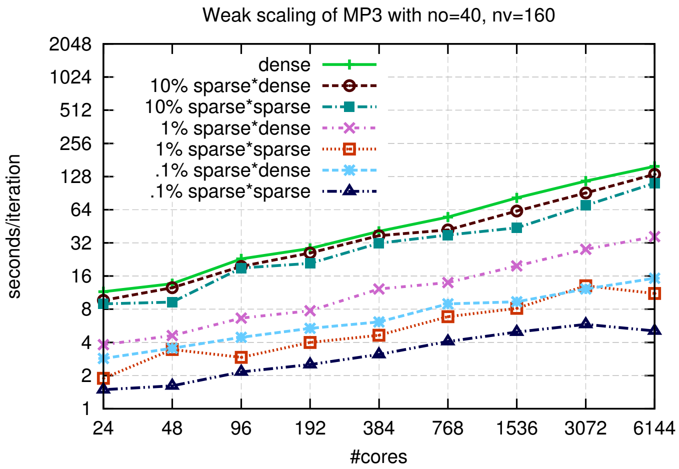
Sparse tensor application: strong scaling

We study the time to solution of the sparse MP3 code, with **(1)** dense *V* and *T* **(2)** sparse *V* and dense *T* **(3)** sparse *V* and *T*



Sparse tensor application: weak scaling

We study the scaling to larger problems of the sparse MP3 code, with **(1)** dense V and T **(2)** sparse V and dense T **(3)** sparse V and T



Data mapping and autotuning

Transitions between contractions require redistribution and refolding

- Base distribution for each tensor
  - default over all processors
  - or user can specify any processor grid mapping
- To contract, tensor is redistributed globally and matricized locally
- Arbitrary sparsity supported via compressed-sparse-row (CSR)
- Performance model used to select best contraction algorithm