Parallel Algorithms across the GraphBLAS Stack

Ariful Azad and Aydın Buluç
Computational Research Division
Lawrence Berkeley National Laboratory (LBNL)

June 28, 2017
ACS HPC and Data Analytics Workshop
Baltimore, MD
Outline of the talk

Part 1: GraphBLAS and Talk Overview

Part 2: Reverse Cuthill-McKee (RCM) Graph Ordering in Distributed-Memory using GraphBLAS

Part 3: Work-Efficient Parallel Sparse Matrix-Sparse Vector Multiplication (SpMSpV) in Shared-Memory
Abstract-- It is our view that the state of the art in constructing a large collection of graph algorithms in terms of linear algebraic operations is mature enough to support the emergence of a standard set of primitive building blocks. This paper is a position paper defining the problem and announcing our intention to launch an open effort to define this standard.
Fast-forward in history

- The idea is older than this SIAM book:
- Several platforms implemented the ideas in the past, such as Star*P
- Current list of active implementations (and version 1.0 of the draft proposal) is available at http://graphblas.org


Today, I will talk about a graph ordering algorithm (RCM) in GraphBLAS and a work-efficient shared-memory algorithm for sparse matrix-sparse vector (SpMSpV) operation in GraphBLAS
The GraphBLAS Stack

- **Traversal Based**
  - connectivity, BFS, independent sets (MIS), graph matching, ordering

- **Centrality**
  - (PageRank, betweenness, closeness)

- **Graph Clustering**
  - (Markov cluster, peer pressure, spectral, local)

- **Shortest Paths**
  - (all-pairs, single-source, temporal)

**GraphBLAS primitives in increasing arithmetic intensity**

- Sparse Matrix-Sparse Vector (SpMSpV)
- Sparse Matrix-Dense Vector (SpMV)
- Sparse Matrix Times Multiple Dense Vectors (SpMM)
- Sparse-Sparse Matrix Product (SpGEMM)
- Sparse-Dense Matrix Product (SpDM³)
GraphBLAS C API Spec (http://graphblasc.org)

- **Goal:** A crucial piece of the GraphBLAS effort is to translate the mathematical specification to an actual Application Programming Interface (API) that
  i. is faithful to the mathematics as much as possible, and
  ii. enables efficient implementations on modern hardware.

- **Impact:** All graph and machine learning algorithms that can be expressed in the language of linear algebra

- **Innovation:** Function signatures (e.g. mxm, vxm, assign, extract), parallelism constructs (blocking v. non-blocking), fundamental objects (masks, matrices, vectors, descriptors), a hierarchy of algebras (functions, monoids, and semiring)

```
GrB_info GrB_mxm(GrB_Matrix *C,
    const GrB_Matrix Mask,
    const GrB_BinaryOp accum,
    const GrB_Semiring op,
    const GrB_Matrix A,
    const GrB_Matrix B
[, const Descriptor desc]);
```

\[ C(-M) \oplus= A^T \oplus.\otimes B^T \]

**A. Buluç, T. Mattson, S. McMillan, J. Moreira, C. Yang.** “Proposal for a GraphBLAS C API” (Working document from the GraphBLAS Signatures Subgroup)
Parallel algorithms for sparse-matrix- sparse matrix multiplication (SpGEMM)

- **Goal:** More scalable SpGEMM algorithms in shared and distributed-memory
- **Applications:** Algebraic multigrid (AMG) restriction, graph computations, quantum chemistry, data mining, interior-point optimization
- **Algorithmic innovations:** (1) Novel shared-memory kernel for in-node parallelism, (2) Split-3D-SpGEMM: an efficient implementation of communication-avoiding SpGEMM
- **Performance:** Split-3D-SpGEMM with new shared-memory kernel (red) beats old state-of-the-art (blue) by 8X at large concurrences

---

An work-efficient parallel algorithm for sparse matrix-sparse vector multiplication (SpMSpV)

- **Goal:** A scalable SpMSpV algorithm without doing more work on higher concurrency
- **Application:** Breadth-first search, graph matching, support vector machines, etc.
- **Algorithmic innovation:**
  - Attains work-efficiency by arranging necessary columns of the matrix into buckets where each bucket is processed by a single thread
  - Avoids synchronization by row-wise partitioning of the matrix on the fly
- **Performance:**
  - First ever work-efficient algorithm for SpMSpV that attains up to 15x speedup on a 24-core Intel Ivy Bridge processor and up to 49x speedup on a 64-core KNL processor
  - Up to an order of magnitude faster than its competitors, especially for sparser vector

A.Azad, A. Buluç. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. IPDPS’17
The Reverse Cuthill-McKee Algorithm in Distributed-Memory

- **Goal:** Find a permutation $P$ of a sparse matrix $A$ so that the bandwidth of $PAP^T$ is small.
- **Application:** Faster iterative solvers, e.g., preconditioned conjugate gradients (PCG).
- **Innovation in Parallel RCM Algorithm:**
  1. **Step1:** level-by-level vertex exploration and ordering. **Approach:** specialized breadth-first search using sparse matrix-sparse vector multiplication (SpMSpV) over a semiring
  2. **Step2:** Ordering of vertices in each level by (parents’ order, degree) pairs. **Approach:** parallel partial sorting.
- **Performance:**
  - First ever distributed-memory RCM algorithm that scales up to 4096 cores on NERSC/Edison.
  - Attains up to 38x speedup on 1028 cores.

A.Azad, M. Jacquelin, A. Buluç, E.Ng. The Reverse Cuthill-McKee Algorithm in Distributed-Memory. IPDPS’17
Outline of the talk

**Part 1:** GraphBLAS and Talk Overview

**Part 2:** Reverse Cuthill-McKee (RCM) Graph Ordering in Distributed-Memory using GraphBLAS

**Part 3:** Work-Efficient Parallel Sparse Matrix-Sparse Vector Multiplication (SpMSpV) in Shared-Memory
Many ways of ordering a matrix

Traditional sparse matrix orderings

Permuting for heavy diagonal (bipartite graph matching)
In this talk, we consider parallel algorithms for reordering sparse matrices.

**Goal:** Find a permutation $P$ so that the bandwidth/profile of $PAP^T$ is small.
Why reordering a matrix

- Better cache reuse in SpMV [Karantasis et al. SC ‘14]
- Faster iterative solvers such as preconditioned conjugate gradients (PCG).

Example: PCG implementation in PETSc

Thermal2 (n=1.2M, nnz=4.9M)
The RCM algorithm

Cuthill-McKee order

Start vertex
(a pseudo-peripheral vertex)

Order vertices by increasing degree

Order vertices by (parents’ order, degree)

Order vertices by parents’ order

Reverse the order of vertices to obtain the RCM ordering
RCM: Challenges in parallelization (in addition to parallelizing BFS)

- Given a start vertex, the algorithm gives a fixed ordering except for tie breaks. **Not parallelization friendly.**

- Unlike traditional BFS, the parent of a vertex is set to a vertex with the minimum label. (i.e., bottom-up BFS is not beneficial)

- Within a level, vertices are labeled by lexicographical order of (parents’ order, degree) pairs, needs sorting
We use **specialized** level-synchronous BFS

Key differences from traditional BFS (Buluç and Madduri, SC ‘11)

1. A parent with smaller label is preferred over another vertex with larger label
2. The labels of parents are passed to their children
3. Lexicographical sorting of vertices in BFS levels

The first two of them are addressed by **sparse matrix-sparse vector multiplication (SpMSpV)** over a semiring

The third challenge is addressed by a lightweight sorting function
Exploring the next-level vertices via SpMSpV

Overload (multiply, add) with (select 2nd, min)

Current frontier

Next frontier

Adjacency matrix

We propagate the parent label information to children during BFS (via semiring multiply)
Ordering vertices via partial sorting

Rules for ordering vertices
1. c and h are ordered before f
2. h is ordered before c

Sort degrees of the siblings many instances of small sortings (avoids expensive parallel sorting)

Current frontier: a, b, c, d, e, f, g, h

Next frontier: h, c, f

Parent’s label: 4, 2, 2, 1
My degree: 4, 2, 1

Sort degrees of the siblings.
Finding a **pseudo peripheral vertex**: Repeated application of the usual BFS (no ordering of vertices within a level).

This is actually quite expensive

- Choose a vertex $u$.
- Among all the vertices that are as far from $u$ as possible, let $v$ be one with minimal degree.
- If $v$ is more eccentric than $u$, then set $u=v$ and repeat previous step, else $v$ is a pseudo-peripheral vertex.

Our SpMSpV is hybrid OpenMP-MPI implementation

- Multithreaded SpMSpV is also fairly complicated and subject to the second half of this talk upcoming slides.
per process. In our hybrid OpenMP-MPI implementation, we found that square grids are close to optimal in practice (within 10% of the best configuration in most cases). When solving an eigenvalue problem \((Ax = \lambda b)\) is solved, the RCM ordering from our distributed-memory algorithm (shown in Figure 3) yields smaller bandwidth.

For four out of eight matrices where SpMP was able to finish within 30 minutes for the runtime of our algorithm with the RCM implementation in SpMP, SpMP did not finish in 30 minutes for 6 cores compared to 24 cores. However, SpMP sometimes loses efficiency due to communication overheads. However, the amount of communication overheads is not shown in Figure 3. These matrices came from a set of real experiments. All matrices, except two, are from the University of Florida sparse matrix collection. Li7Nmax6 and Nm7 [22] are from nuclear configuration interaction calculations.

Structural information on the sparse matrices used in our experiments. All matrices, except two, are from the University of Florida sparse matrix collection. Li7Nmax6 and Nm7 [22] are from nuclear configuration interaction calculations.
Results: Scalability on NERSC/Edison
(6 threads per MPI process)

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>54</td>
<td>54</td>
</tr>
<tr>
<td>216</td>
<td>216</td>
</tr>
<tr>
<td>1,014</td>
<td>1,014</td>
</tr>
<tr>
<td>4,056</td>
<td>4,056</td>
</tr>
</tbody>
</table>

#vertices: 1.1M  #edges: 89M
Bandwidth before: 1,036,475  after: 23,813

Communication dominates

[Graph showing scalability results]

Bandwidth before: 1,036,475
Bandwidth after: 23,813

Communication dominates by 30x
Scalability on NERSC/Edison
(6 threads per MPI process)

#vertices: 78M  
Bandwidth before: 14,169,841  
#edges: 760M  
Bandwidth after: 361,755

Larger graphs continue scaling
Why is this significant?

- We managed to scaling RCM in distributed memory relatively easily.

- The community should pad themselves on the back

- Research in BFS (e.g. Graph500) and graph primitives (GraphBLAS) made this possible.

- Flashback to 1995 [Barnard, Pothen, Simon]:

  “The spectral envelope-reduction algorithm has several features which set it apart from the earlier reordering algorithms such as the GPS, GK, or RCM algorithms. These algorithms employ local-search in the adjacency graph of the matrix. All of them try to find a pseudo-diameter in the graph by generating a long level-structure by breadth first-search beginning from a suitable vertex. These types of algorithms generally do not vectorize, and there is no obvious way to implement them in parallel.”
Outline of the talk

**Part 1:** GraphBLAS and Talk Overview

**Part 2:** Reverse Cuthill-McKee (RCM) Graph Ordering in Distributed-Memory using GraphBLAS

**Part 3:** Work-Efficient Parallel Sparse Matrix-Sparse Vector Multiplication (SpMSpV) in Shared-Memory
A lower bound for $\text{SpMSpV}$

Considering Erdos-Renyi graph $G(n, d/n)$

**Lower bound of SpMSpV:** $df$ (no matrix/vector dimension)
SpMSpV via sparse accumulator (SPA)

Can be done in \(O(df)\) time; attains lower bound

We parallelize this algorithm
Shared-memory parallelization of SpMSpV (row split)

Explicitly split local submatrices into t (#threads) pieces

<table>
<thead>
<tr>
<th>Work efficient?</th>
<th>Synchronization needed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>No : O(tf + df ) total work</td>
<td>No</td>
</tr>
</tbody>
</table>
Shared-memory parallelization of SpMSpV (*column split*)

Space complexity: $O(n)$ storage per thread

<table>
<thead>
<tr>
<th>Work efficient?</th>
<th>Synchronization needed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Yes</strong>: $O(df)$ total work</td>
<td><strong>Yes (in merging)</strong></td>
</tr>
</tbody>
</table>
A work-efficient and synchronization-avoiding SpMSpV algorithm using buckets

- **Multi-step algorithm**: keep good features of both row-split and column-split algorithms (*SpMSpV-bucket*)

- **Step1**: Arrange columns in buckets. Each bucket stores consecutive row indices. [similar to the column-split algorithm]

![Diagram]

**Complexity:** $O(df)$

- $A$ (CSC format)

- **Step 1 (bucketing)**
  - Thread 1
    - $(2, •)$, $(5, •)$, $(7, •)$
  - Thread 2

- **Buckets**
  - B1: $(0, s’), (0, e’)$
  - B2: $(3, f’), (2, p’)$
  - B3: $(5, g’), (4, q’), (4, t’)$
  - B4: $(6, h’)$

- **Multiplied values**
Step 2: Merge each bucket independently by a thread. [similar to the row-split algorithm]

Complexity: $O(df)$
A work-efficient and synchronization-avoiding SpMSpV algorithm

- **Step 3:** Concatenate entries to the result vector

Complexity: $O(\text{nnz}(y))$
Smaller than $O(df)$
### SpMSpV-bucket algorithm

<table>
<thead>
<tr>
<th>Work efficient?</th>
<th>Synchronization needed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes: $O(df)$ total work</td>
<td>No (in each step)</td>
</tr>
<tr>
<td>At most $3df$ work</td>
<td></td>
</tr>
</tbody>
</table>

- **Other tricks for practical performance**
  - Load balancing: multiple buckets per thread
  - Cache efficiency: small thread-private buffers filled up first before writing to buckets
Relative performance of SpMSPv algorithms

Graph: ljournal-2008  Vertices: 5M, Edges: 78M

(b) 12 threads on Edison

An order of magnitude faster for very sparse vectors
Strong scaling of SpMSpV algorithms on Edison

SpMSpV when used in BFS: varying sparsity of the input vector
Up to 4x faster than the second best algorithm
Algorithmic innovation:
- First-ever work-efficient SpMSpV algorithm
- Attains work-efficiency by arranging necessary columns of the matrix into buckets
- Avoids synchronization by processing buckets independently

Impact:
- Up to an order of magnitude faster than state-of-the-art algorithms when the input vector is very sparse
- Will expedite a large class of graph and machine learning algorithms
Thanks for your attention