The Reverse Cuthill-McKee Algorithm in Distributed-Memory

Ariful Azad
Lawrence Berkeley National Laboratory (LBNL)

SIAM CSC 2016, Albuquerque

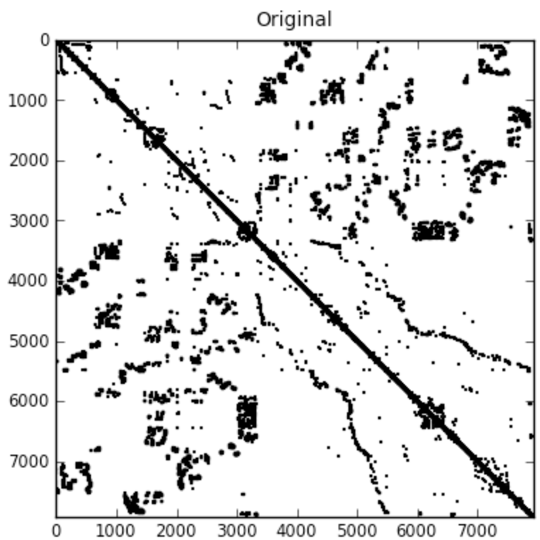
Acknowledgements

- Joint work with
  - Aydın Buluç
  - Mathias Jacquelin
  - Esmond Ng

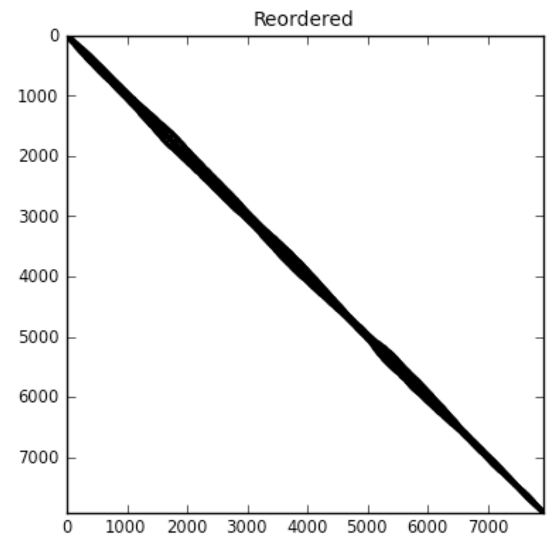
- Funding
  - DOE Office of Science
  - Time allocation at the DOE NERSC Center

Reordering a sparse matrix

- In this talk, I consider parallel algorithms for **reordering** sparse matrices
- **Goal:** Find a permutation P so that the bandwidth/ profile of PAP^T is small.



Before permutation

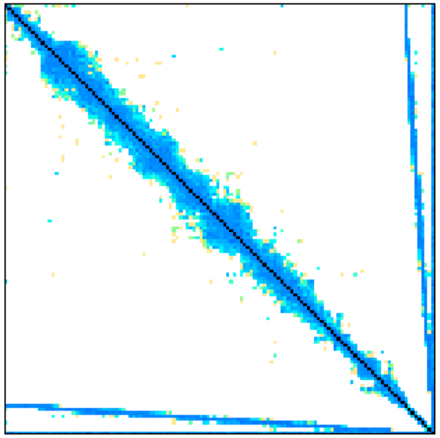


After permutation

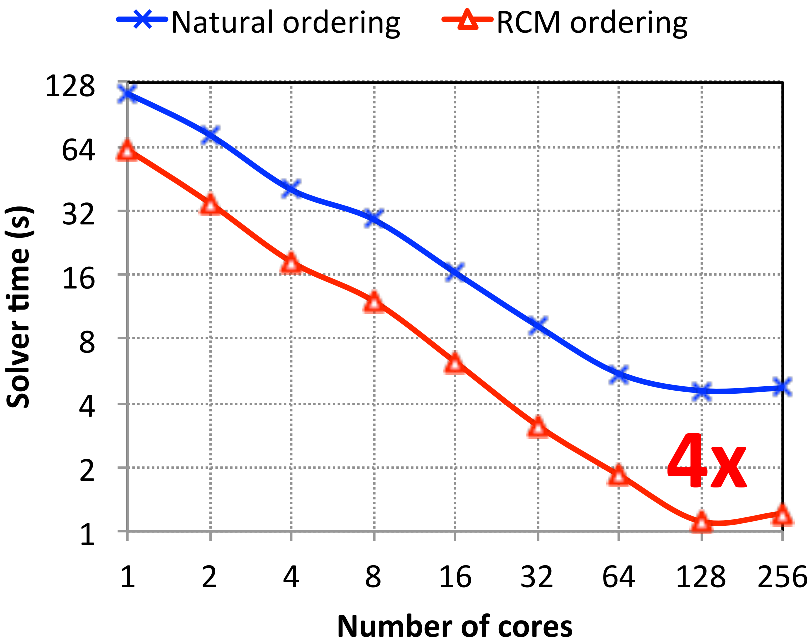
Why reordering a matrix

- 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)



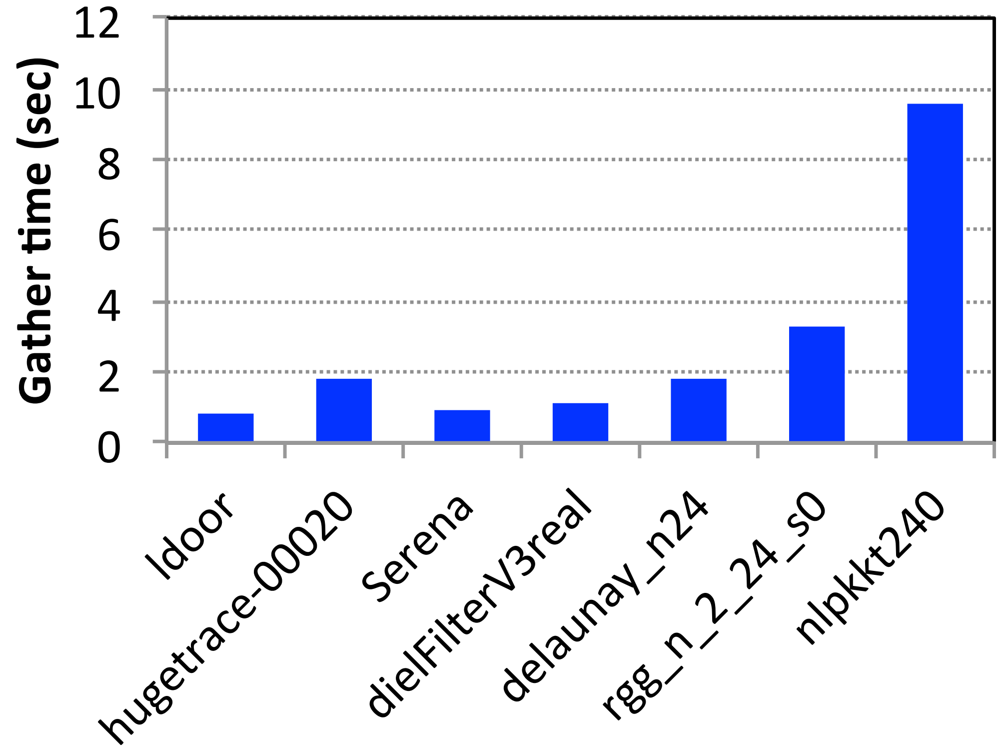
The case for the Reverse Cuthill-McKee (RCM) algorithm

- Finding a permutation to minimize the bandwidth is NP-complete. [Papadimitriou '76]
- Heuristics are used in practice
  - Examples: the Reverse Cuthill-McKee algorithm, Sloan's algorithm
- We focus on the Reverse Cuthill-McKee (RCM) algorithm
  - Simple to state
  - Easy to understand
  - Relatively easy to parallelize

The case for distributed-memory algorithm

- Enable solving very large problems
- More practical: The matrix is already distributed
  - gathering the distributed matrix onto a node for serial execution is expensive.

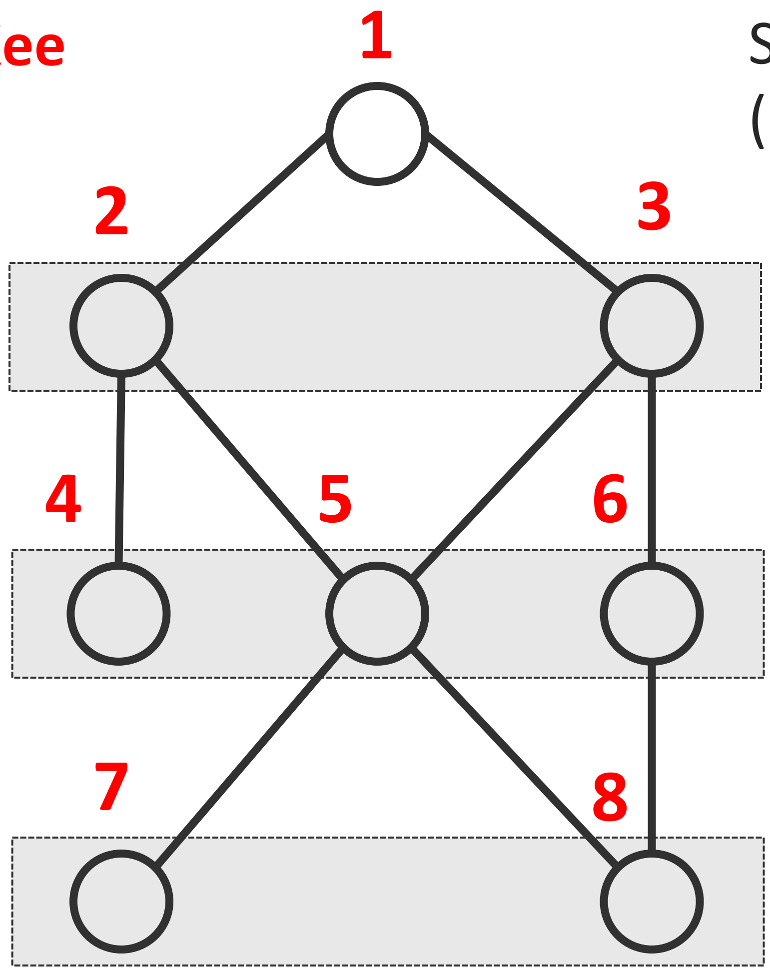
Time to gather a graph on a node from 45 nodes of NERSC/Edison (Cray XC30)



Distributed algorithms are cheaper and scalable

The RCM algorithm

Cuthill-McKee order



Start vertex
(a pseudo-peripheral vertex)

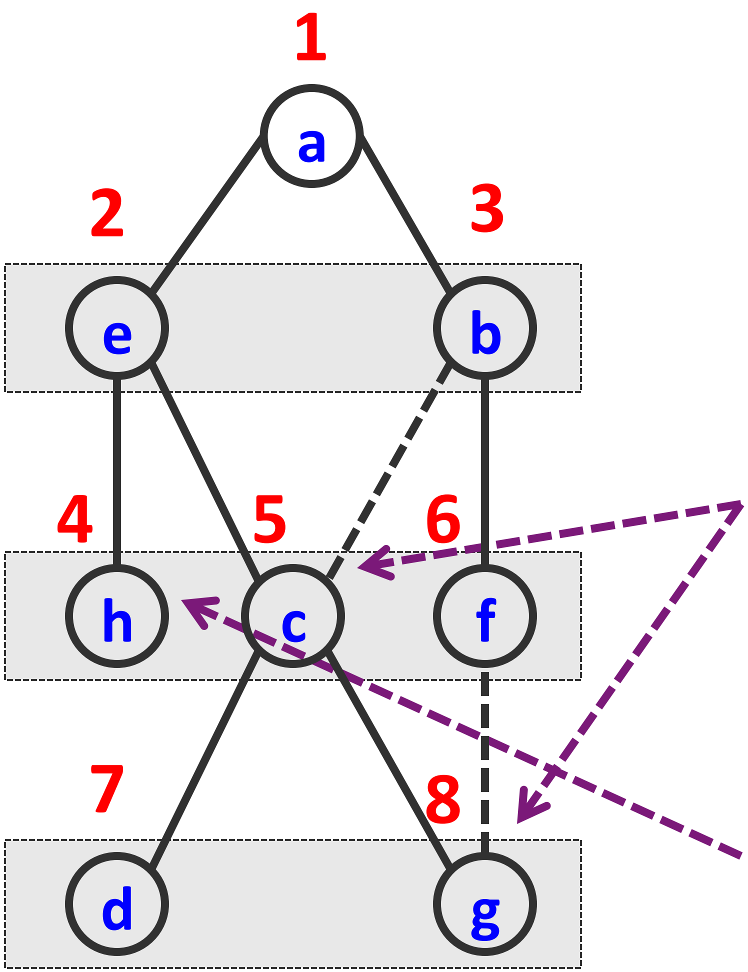
Order vertices by increasing degree

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

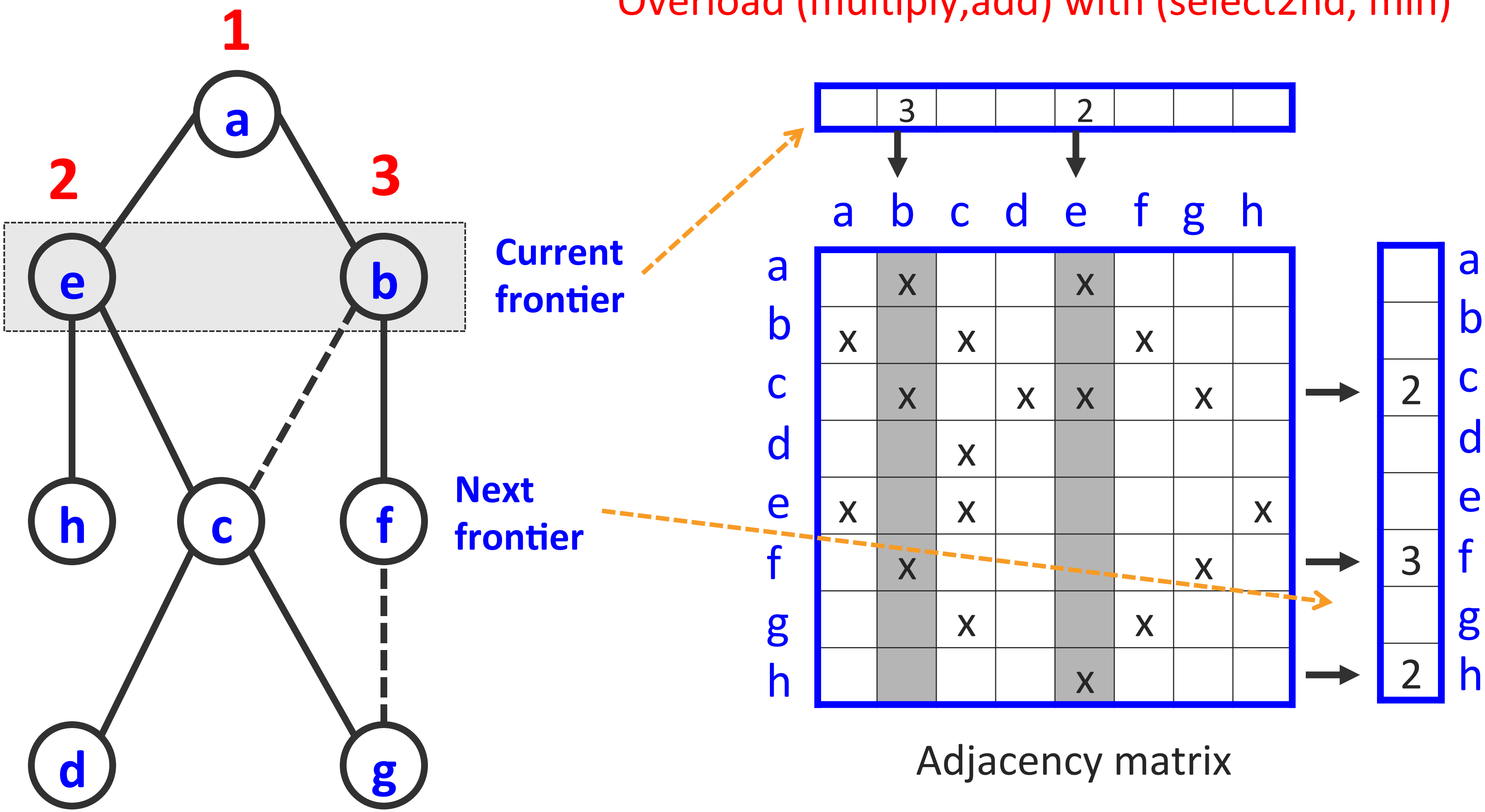
Our approach to address parallelization challenges

- 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

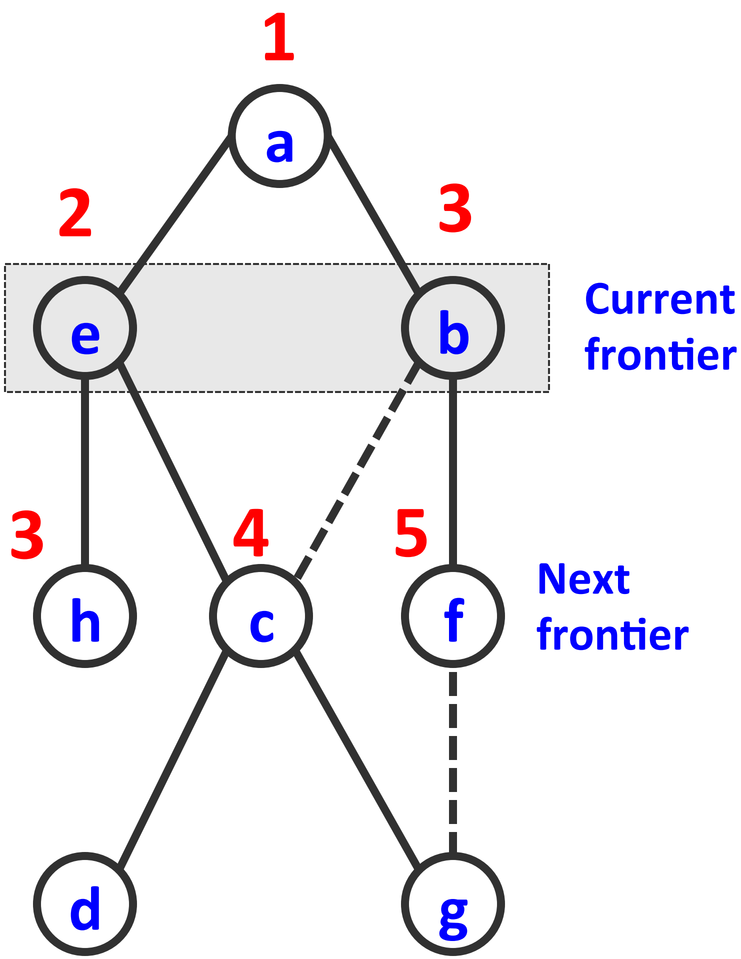
- The first two of them are addressed by sparse matrix-sparse vector multiplication (SpMSpV) over a semiring
- The third challenge is addressed by a lightweight sorting function

Exploring the next-level vertices via SpMSpV

Overload (multiply,add) with (select2nd, min)



Ordering vertices via partial sorting



Sort degrees of the siblings

many instances of small sortings

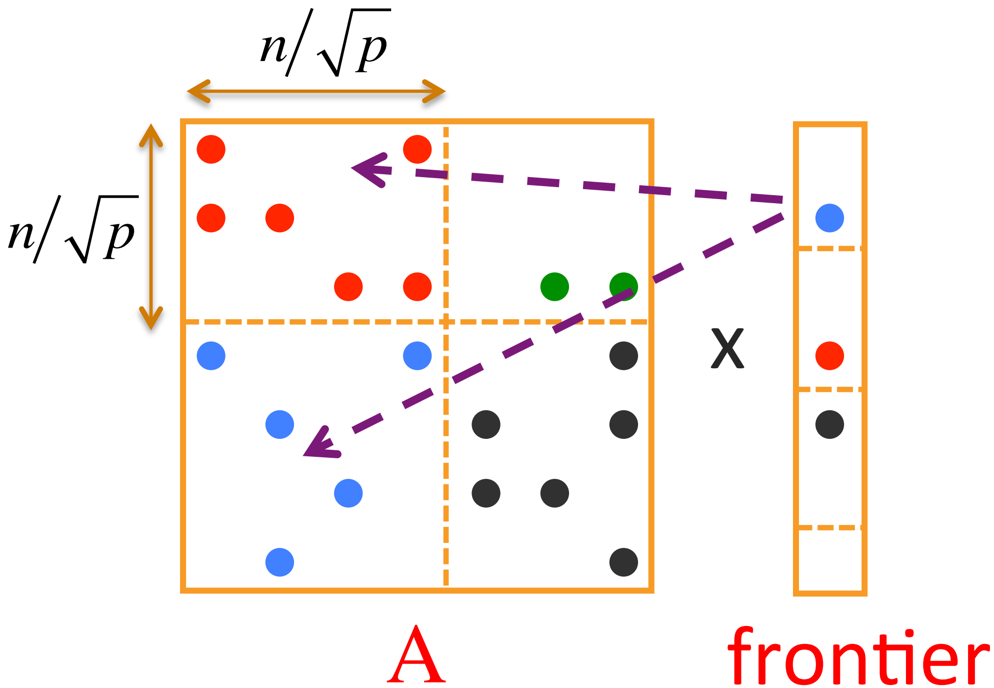
(avoids expensive parallel sorting)

	a	b	c	d	e	f	g	h
Parent's label			2			3		2
My degree			4			2		1

Rules for ordering vertices

1. c and h are ordered before f
2. h is ordered before c

Distributed memory parallelization (SpMSpV)



P processors are arranged in $\sqrt{p} \times \sqrt{p}$ Processor grid

ALGORITHM:

1. Gather vertices in *processor column* **[communication]**
2. Local multiplication [computation]
3. Find owners of the current frontier's adjacency and exchange adjacencies in *processor row* **[communication]**

Distributed-memory partial sorting

- Bin vertices by their parents' labels
  - All vertices in a bin is assigned to a single node
  - Needs AllToAll communication
- Sequentially sort the degree of vertices in a single node

Computation and communication complexity

Operation	Per processor **Computation** (lower bound)	Per processor **Comm (latency)**	Per processor **Comm (bandwidth)**
SpMSpV	$\frac{m}{p}$	$diameter * \alpha\sqrt{p}$	$\beta\left(\frac{m}{p}+\frac{n}{\sqrt{p}}\right)$
Sorting	$\frac{n}{p}\log(n/p)$	$diameter * \alpha p$	$\beta\frac{n}{p}$

n: number of vertices, m: number of edges

α : latency (0.25 µs to 3.7 µs MPI latency on Edison)
β : inverse bandwidth (~8GB/sec MPI bandwidth on Edison)
p : number of processors

Other aspects of the algorithm

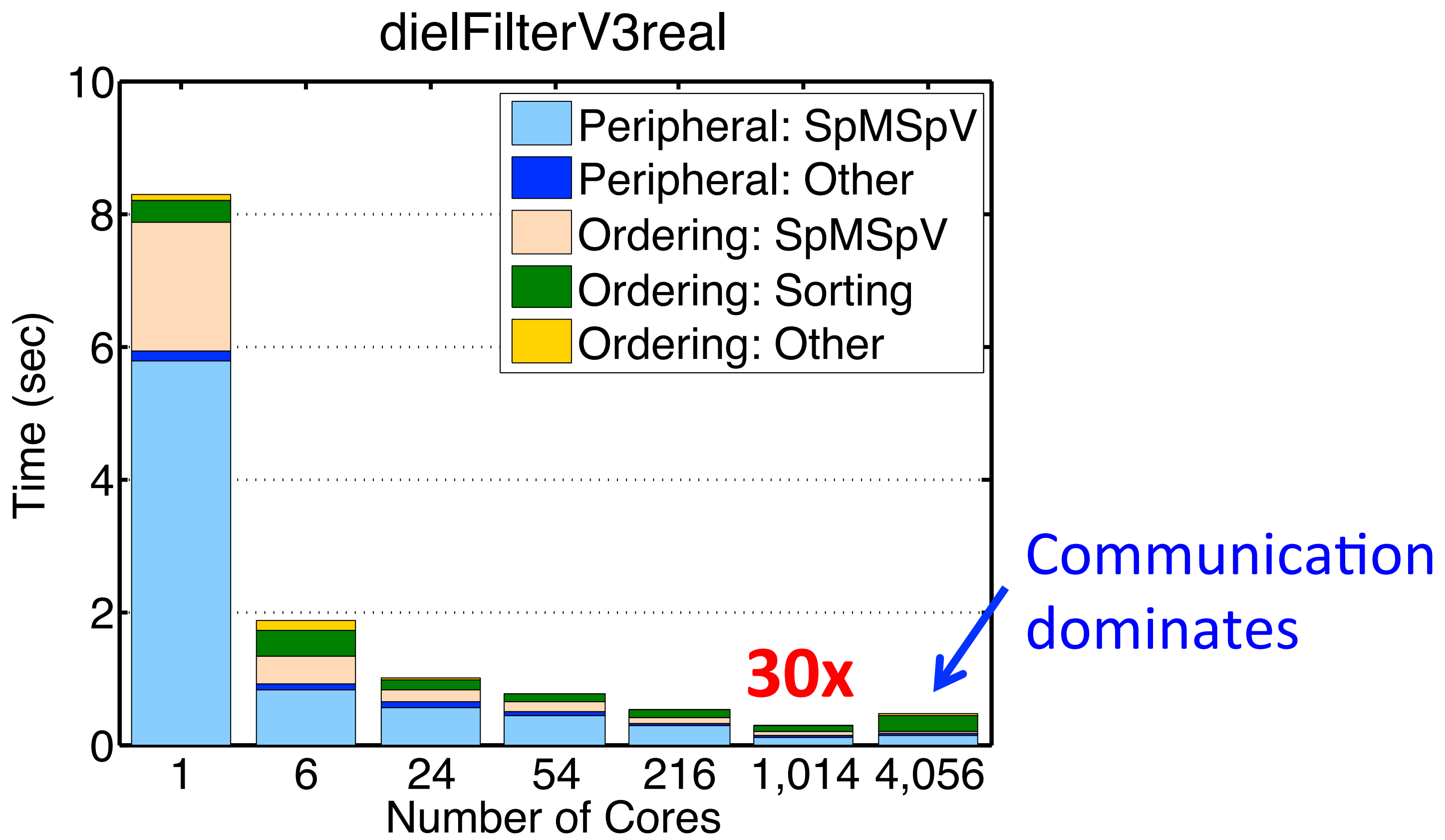
- Finding a pseudo peripheral vertex.
  - Repeated application of the usual BFS (no ordering of vertices within a level)

- Our SpMSpV is hybrid OpenMP-MPI implementation
  - Multithreaded SpMSpV is also fairly complicated and subject to another work

Results: Scalability on NERSC/Edison (6 threads per MPI process)

#vertices: 1.1M #edges: 89M

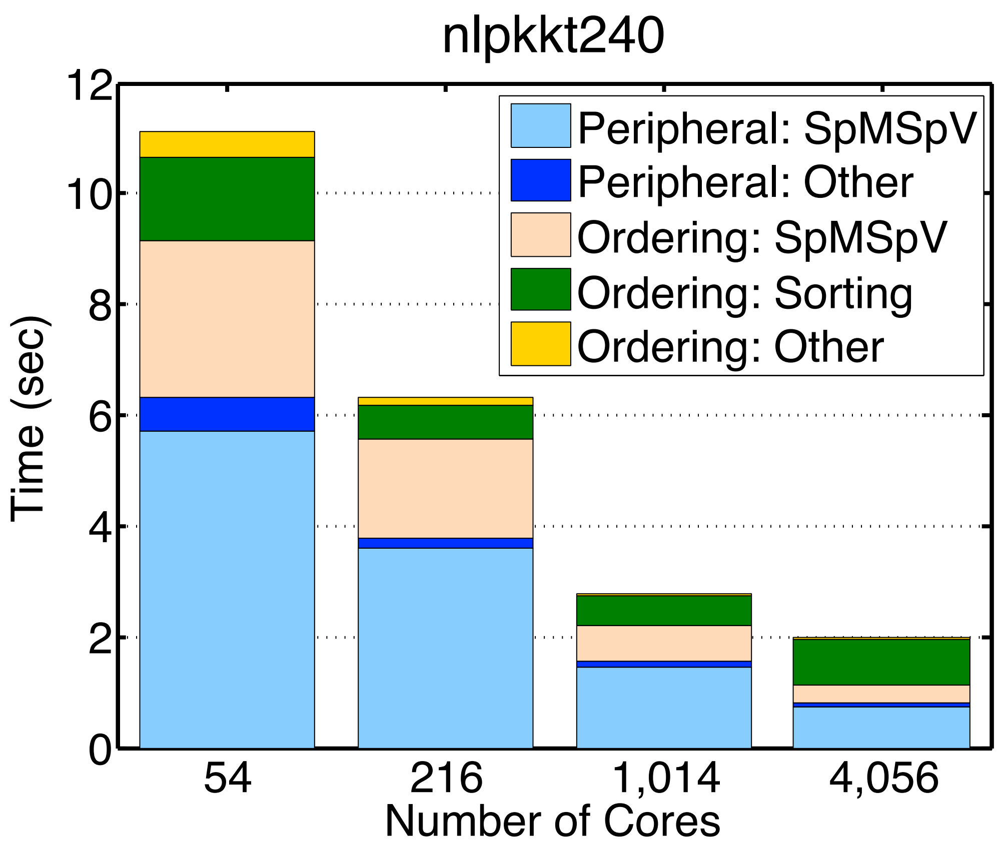
Bandwidth before: 1,036,475 after: 23,813



Scalability on NERSC/Edison (6 threads per MPI process)

#vertices: 78M #edges: 760M

Bandwidth before: 14,169,841 after: 361,755

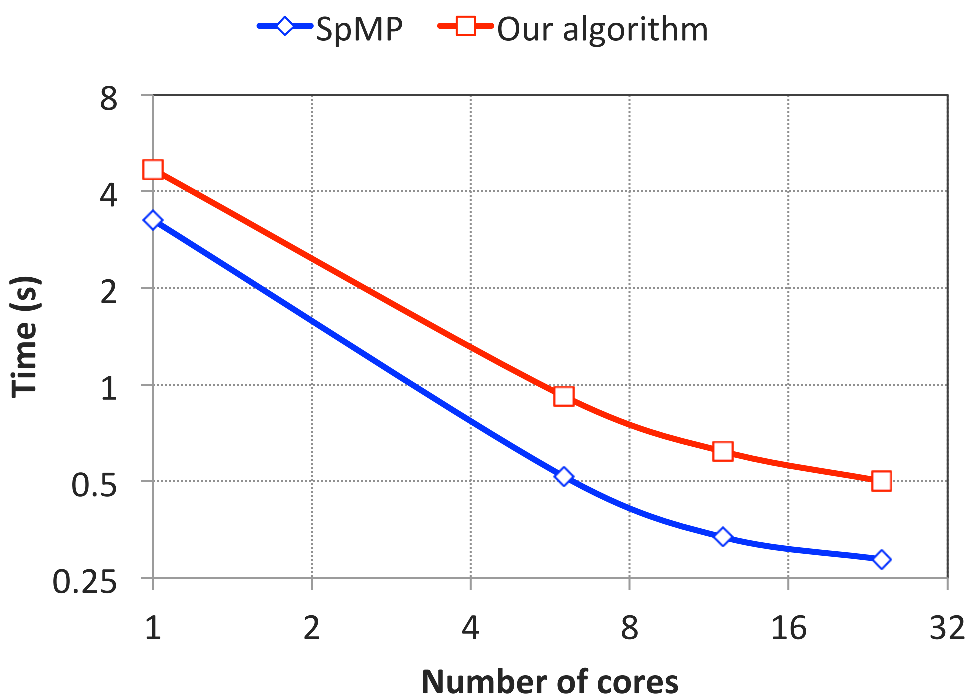


Larger graphs continue scaling

Single node performance
NERSC/Edison (2x12 cores)

- SpMP (Sparse Matrix Pre-processing) package by Park et al. (https://github.com/jspark1105/SpMP)
- We switch to MPI+OpenMP after 12 cores

Matrix: ldoor #vertices: 1M #edges: 42M



If the matrix is already Distributed in 1K cores (~45 nodes)

Time to gather: 0.82 s

making the distributed algorithm more profitable

Conclusions

- For many practical problems, the RCM ordering expedites iterative solvers
- No scalable distributed memory algorithm for RCM ordering exists
  - forcing us gathering an already distributed matrix on a node and use serial algorithm (e.g., in PETSc), which is expensive
- We developed a distributed-memory RCM algorithm using SpMSpV and partial sorting
- The algorithm scales up to 1K cores on modern supercomputers.

Thanks for your attention