Scaling up graph algorithms on emerging multicore systems

Kamesh Madduri

KMadduri@lbl.gov

Graphs are pervasive in large-scale data analysis

Sources of massive data: petascale simulations, experimental devices, the Internet, scientific applications.
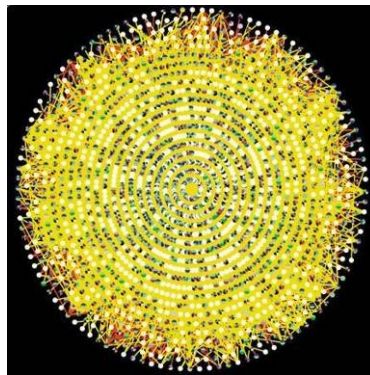
Cosmology
Application: Outlier detection.
Challenges: petascale datasets.
Graph problems: clustering, matching.

Bioinformatics
Application: Identifying drug target proteins.
Challenges: Data heterogeneity, quality.
Graph problems: centrality, clustering.

Social Informatics
Application: Discover emergent communities, model spread of information.
Challenges: new analytics routines, uncertainty in data.
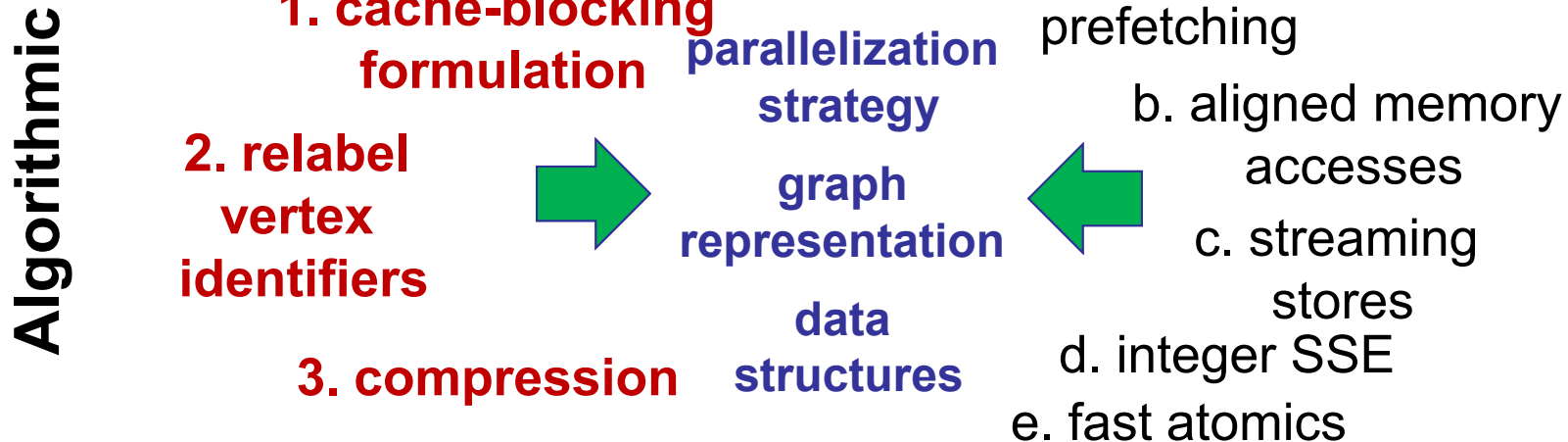Graph problems: clustering, shortest paths, flows.
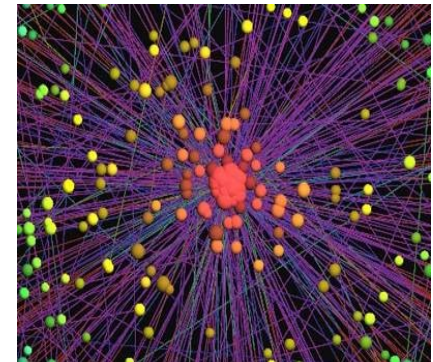
New challenges for analysis: data sizes, heterogeneity, uncertainty, data quality, and dynamic/temporal nature of data.

Talk Outline

Strategies to speed up graph-traversal based algorithms on current and emerging cache-based multicore systems.

- A closer look at parallel Breadth-First Search (BFS) on current systems.

- The techniques and their applicability:

Algorithmic

1. cache-blocking formulation

2. relabel vertex identifiers

3. compression

parallelization strategy

graph representation

data structures

a. software prefetching

b. aligned memory accesses

c. streaming stores

d. integer SSE

e. fast atomics

Arch. specific

The problems: #1. The locality challenge
"Large memory footprint, low spatial and temporal locality impede performance"

Serial Performance of "approximate betweenness centrality" on a 2.67 GHz Intel Xeon 5560 (12 GB RAM, 8MB L3 cache)

Input: Synthetic R-MAT graphs (# of edges $m = 8n$)

~ 5X drop in performance
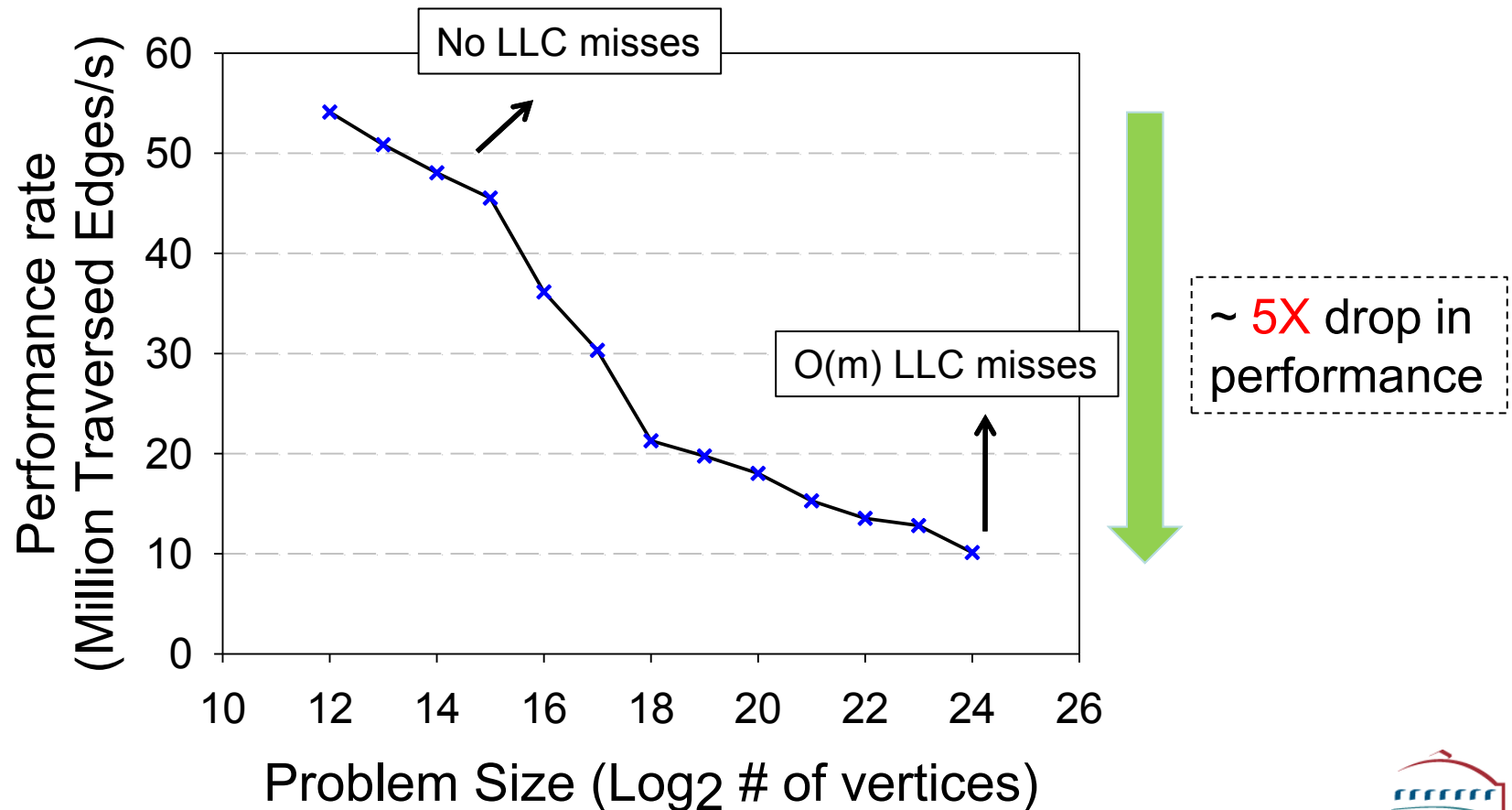
The problems: #2. The parallel scaling challenge
"Classical parallel graph algorithms perform poorly on current parallel systems"

- Graph topology assumptions in classical algorithms do not match real-world datasets

- Parallelization strategies at loggerheads with techniques for enhancing memory locality

- Classical "work-efficient" graph algorithms may not fully exploit new architectural features

  - Increasing complexity of memory hierarchy (x86), DMA support (Cell), wide SIMD, floating point-centric cores (GPUs).

- Tuning implementation to minimize parallel overhead is non-trivial

  - Shared memory: minimizing overhead of locks, barriers.

  - Distributed memory: bounding message buffer sizes, bundling messages, overlapping communication w/ computation.

This talk: Parallel BFS performance on cache-based multicore platforms

- Minimize execution time on current systems.
- Identify scalable parallelization strategies for multi-socket, multicore shared memory systems.

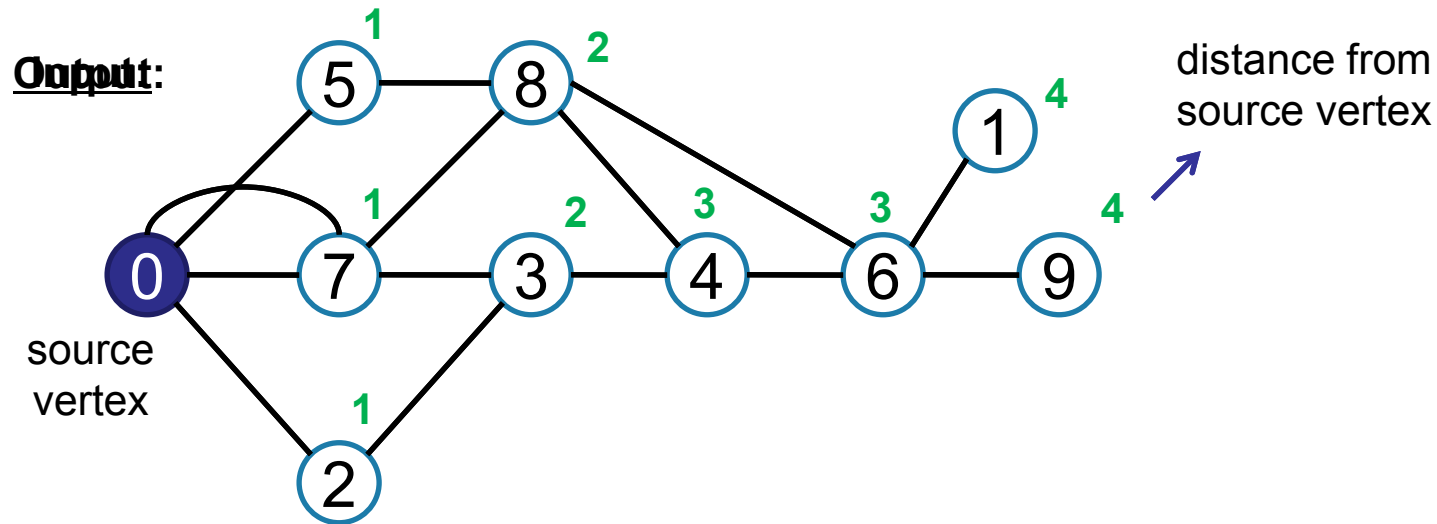| Problem Spec.	Assumptions for this talk
No. of vertices/edges	$10^6 \sim 10^9$
Edge/vertex ratio	1 ~ 100
Static/dynamic?	Static
Diameter	O(1) ~ O(log n)
Weighted/Unweighted	Unweighted
Vertex degree distribution	Unbalanced ("power law")
Directed/undirected?	Both
Simple/multi/hypergraph?	Multigraph
Granularity of computation at vertices/edges?	Minimal
Exploiting domain-specific characteristics?	Partially

Test data

orkut beta

LiveJournal

You Tube
Broadcast Yourself

flickr BETA ™

(Data: Mislove et al., IMC 2007.)

Synthetic R-MAT networks

BERKELEY LAB

Graph traversal (BFS) problem definition

Output:

5 1

8 2

1 4

0

7 1

3 2

4 3

6 3

9 4

source vertex

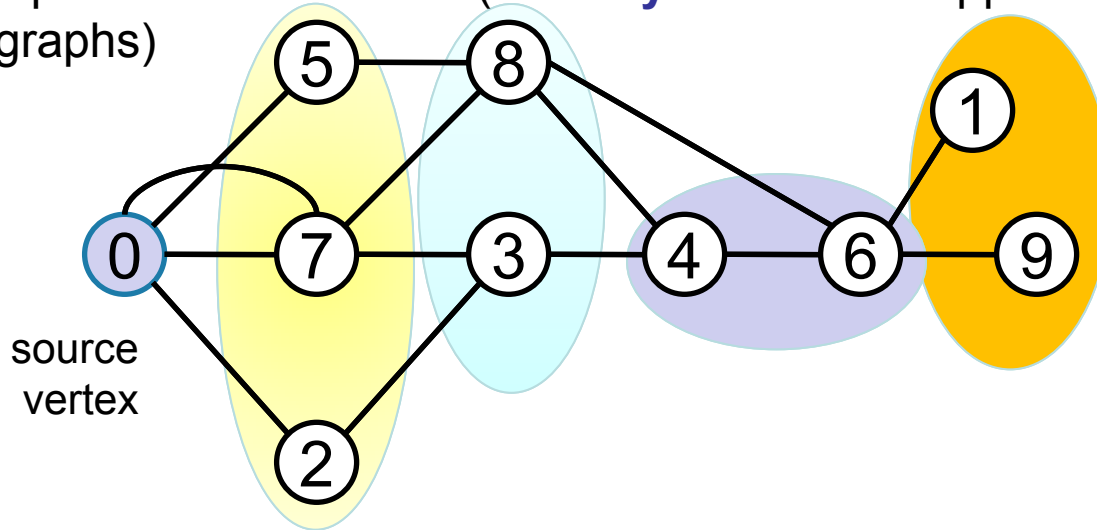**2** 1

distance from source vertex

Memory requirements (# of machine words):
- Sparse graph representation: m+n
- Stack of visited vertices: n
- Distance array: n

Parallel BFS Strategies

1. Expand current frontier (**level-synchronous** approach, suited for **low diameter** graphs)

- O(D) parallel steps
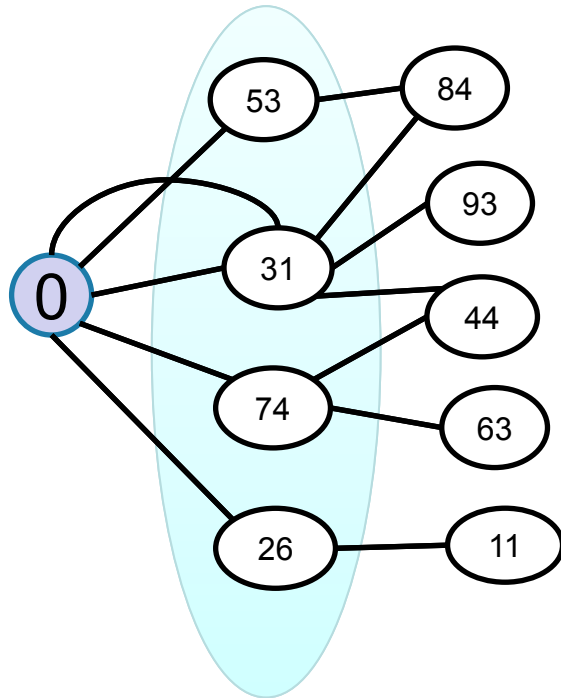- Adjacencies of all vertices in current frontier are visited in parallel

source vertex

2. Stitch multiple concurrent traversals (Ullman-Yannakakis approach, suited for **high-diameter** graphs)
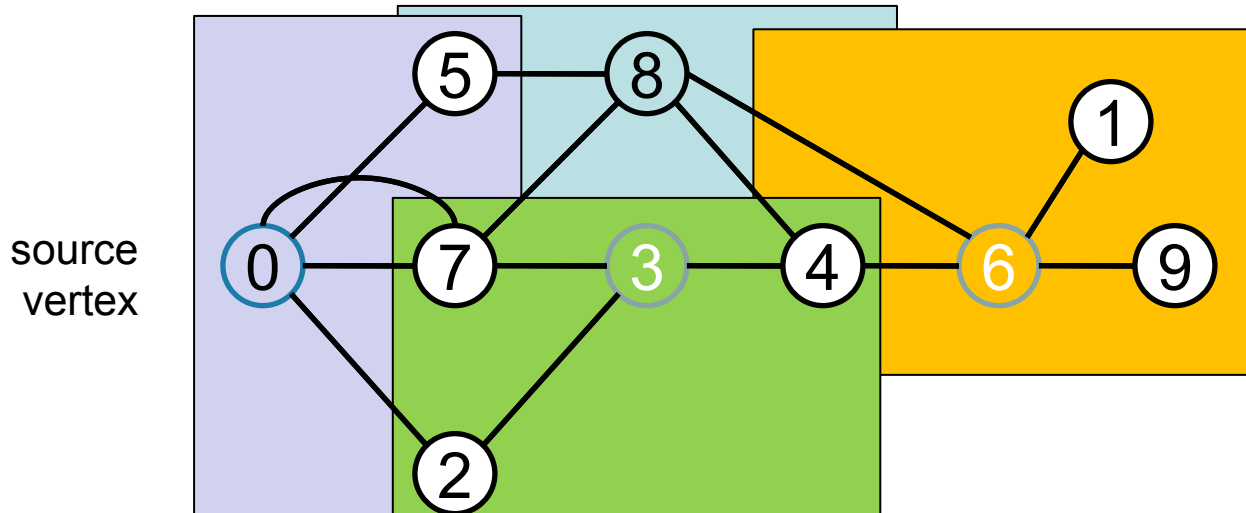
- path-limited searches from "super vertices"
- APSP between "super vertices"

source vertex

A deeper dive into the "level synchronous" strategy

Locality (where are the random accesses originating from?)


1. Ordering of vertices in the "current frontier" array, i.e., accesses to adjacency indexing array, cumulative accesses O(n).
2. Ordering of adjacency list of each vertex, cumulative O(m).
3. Sifting through adjacencies to check whether visited or not, cumulative accesses O(m).
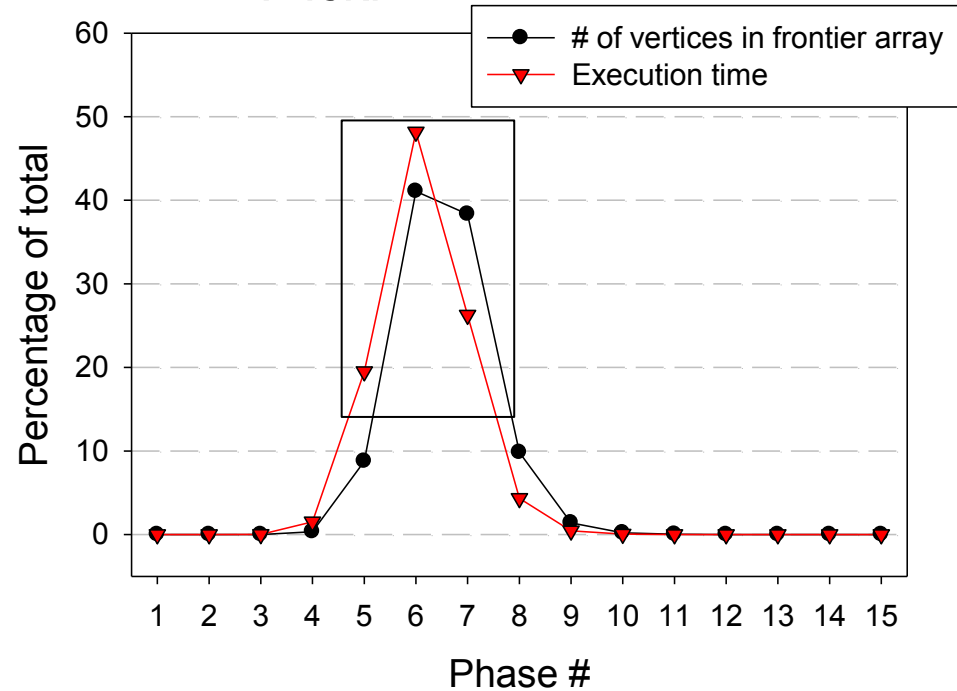
1. Access Pattern: idx array -- 53, 31, 74, 26
2,3. Access Pattern: d array -- 0, 84, 0, 84, 93, 44, 63, 0, 0, 11
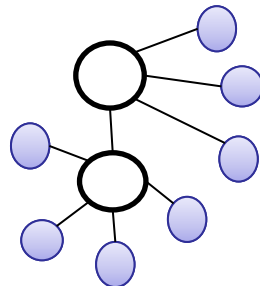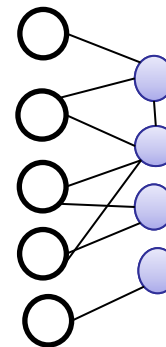
Performance Observations
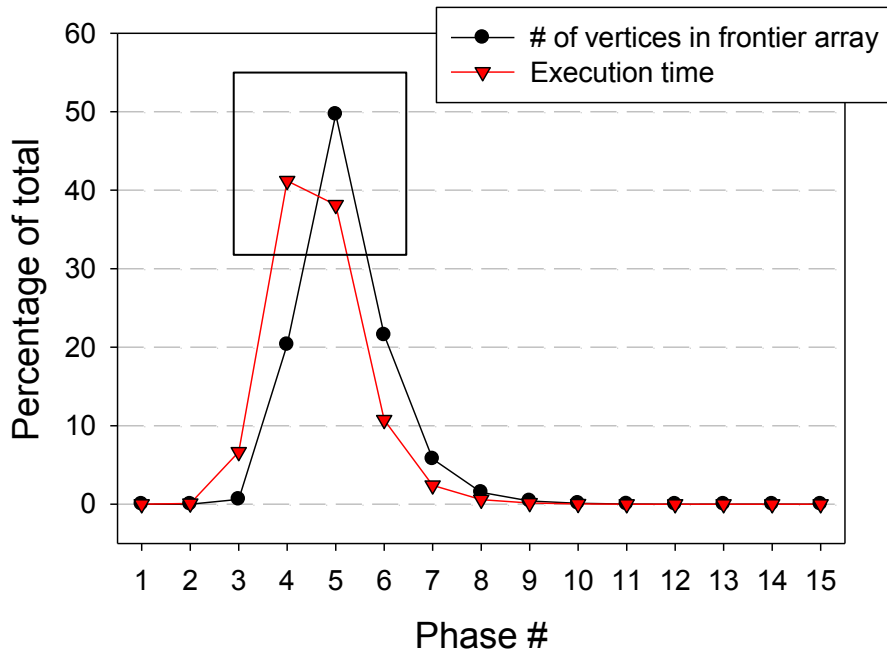
Improving locality: Vertex relabeling
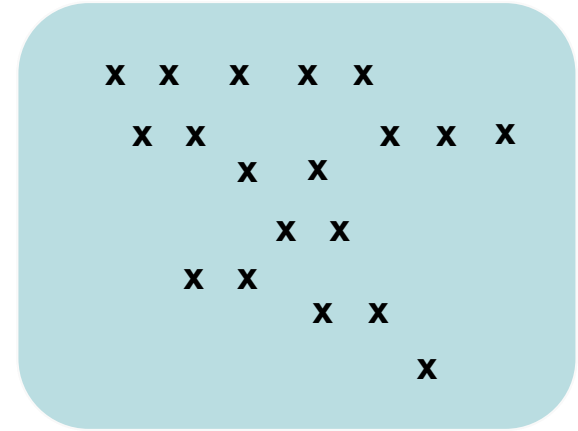
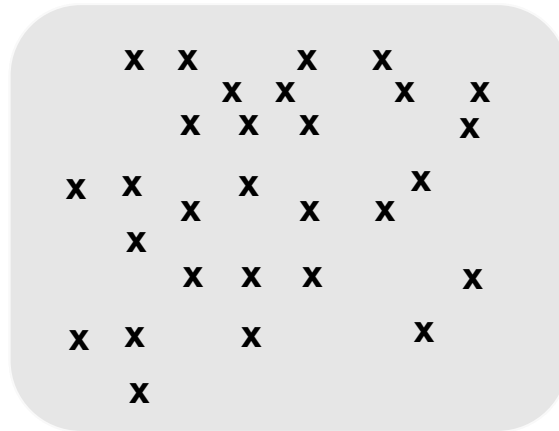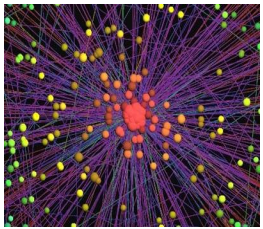- Well-studied problem, slight differences in problem formulations
  - Linear algebra: sparse matrix column reordering to reduce bandwidth, reveal dense blocks.
  - Databases/data mining: reordering bitmap indices for better compression; permuting vertices of WWW snapshots, online social networks for compression
- NP-hard problem, several known heuristics
  - We require fast, linear-work approaches
  - Existing ones: BFS or DFS-based, Cuthill-McKee, Reverse Cuthill-McKee, exploit overlap in adjacency lists, dimensionality reduction
  - Yet another heuristic, coming up …

Improving locality: Optimizations

- Recall: Potential O(m) non-contiguous memory references in edge traversal (to check if vertex is visited).
  - e.g., access order: 53, 31, 31, 26, 74, 84, 0, …
- Objective: Reduce TLB misses, private cache misses, exploit shared cache.

- Optimizations:
  1. Sort the adjacency lists of each vertex – helps order memory accesses, reduce TLB misses.
  2. Permute vertex labels – enhance spatial locality.
  3. Cache-blocked edge visits – exploit temporal locality.
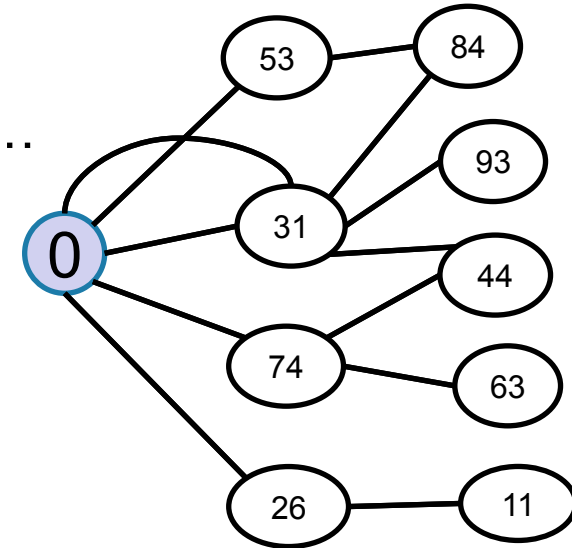
Improving locality: Cache blocking

linear processing

New: cache-blocked approach
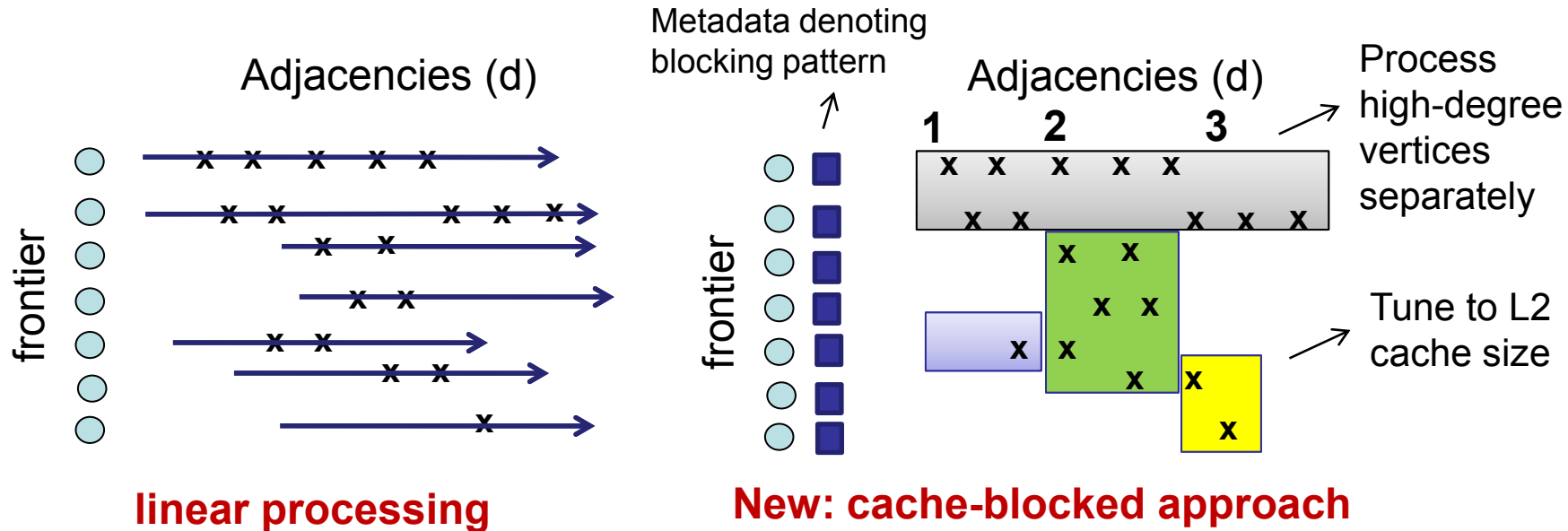
- Instead of processing adjacencies of each vertex serially, exploit sorted adjacency list structure w/ blocked accesses

- Requires multiple passes through the frontier array, tuning for optimal block size.

  - Note: frontier array size may be O(n)

Vertex relabeling heuristic

Similar to older heuristics, but tuned for small-world networks:

1. High percentage of vertices with (out) degrees 0, 1, and 2 in social and information networks => store adjacencies explicitly (in indexing data structure).
   - Augment the adjacency indexing data structure (w/ two additional words) and frontier array (w/ one word)

2. Process "high-degree vertices" adjacencies in linear order, but other vertices with d-array cache blocking.

3. Form dense blocks around high-degree vertices
   - Reverse Cuthill-McKee, removing degree 1 and degree 2 vertices

Architecture-specific Optimizations

1. Software prefetching on the Intel Core i7 (supports 32 loads and 20 stores in flight)
   - Speculative loads of index array and adjacencies of frontier vertices will reduce compulsory cache misses.
   - Hardware prefetcher doesn't help, disable it.
2. Aligning adjacency lists to optimize memory accesses
   - 16-byte aligned loads and stores are faster.
   - Alignment helps reduce cache misses due to fragmentation
   - 16-byte aligned non-temporal stores (during creation of new frontier) are fast.
3. SIMD SSE integer intrinsics to process "high-degree vertex" adjacencies.
4. Fast atomics (BFS is lock-free w/ low contention, and CAS-based intrinsics have very low overhead)
   - Pipelined atomics in the near future
5. Hugepage support (significant TLB miss reduction)
6. NUMA-aware memory allocation exploiting first-touch policy

Experimental Setup

Network	n	m	Max. out-degree	% of vertices w/ out-degree 0,1,2
Orkut	3.07M	223M	32K	5
LiveJournal	5.28M	77.4M	9K	40
Flickr	1.86M	22.6M	26K	73
Youtube	1.15M	4.94M	28K	76
R-MAT	8M-64M	8n	$n^{0.6}$	

Intel Xeon 5560 (Core i7, "Nehalem")
- 2 sockets x 4 cores x 2-way SMT
- 12 GB DRAM, 8 MB shared L3
- 51.2 GBytes/sec peak bandwidth
- 2.66 GHz proc.

Performance averaged over
10 different source vertices, 3 runs each.
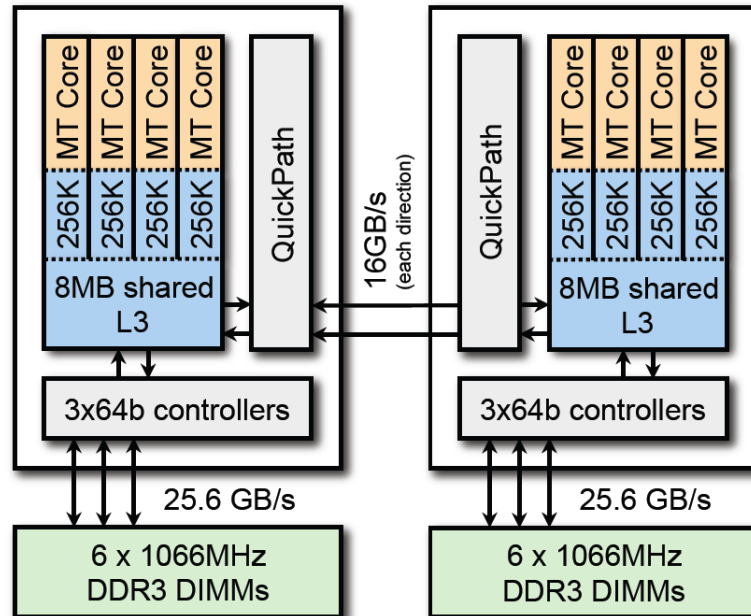
Impact of optimization strategies

Optimization	Generality	Impact*	Tuning required?
(Preproc.) Sort adjacency lists	High	--	No
(Preproc.) Permute vertex labels	Medium	--	Yes
Preproc. + binning frontier vertices + cache blocking	M	**2.5x**	Yes
Lock-free parallelization	M	2.0x	No
Low-degree vertex filtering	Low	1.3x	No
Software Prefetching	M	1.10x	Yes
Aligning adjacencies, streaming stores	M	1.15x	No
Fast atomic intrinsics	H	**2.2x**	No

* Optimization speedup (performance on 4 cores) w.r.t baseline parallel approach, on a synthetic R-MAT graph ($n=2^{23}, m=2^{26}$)
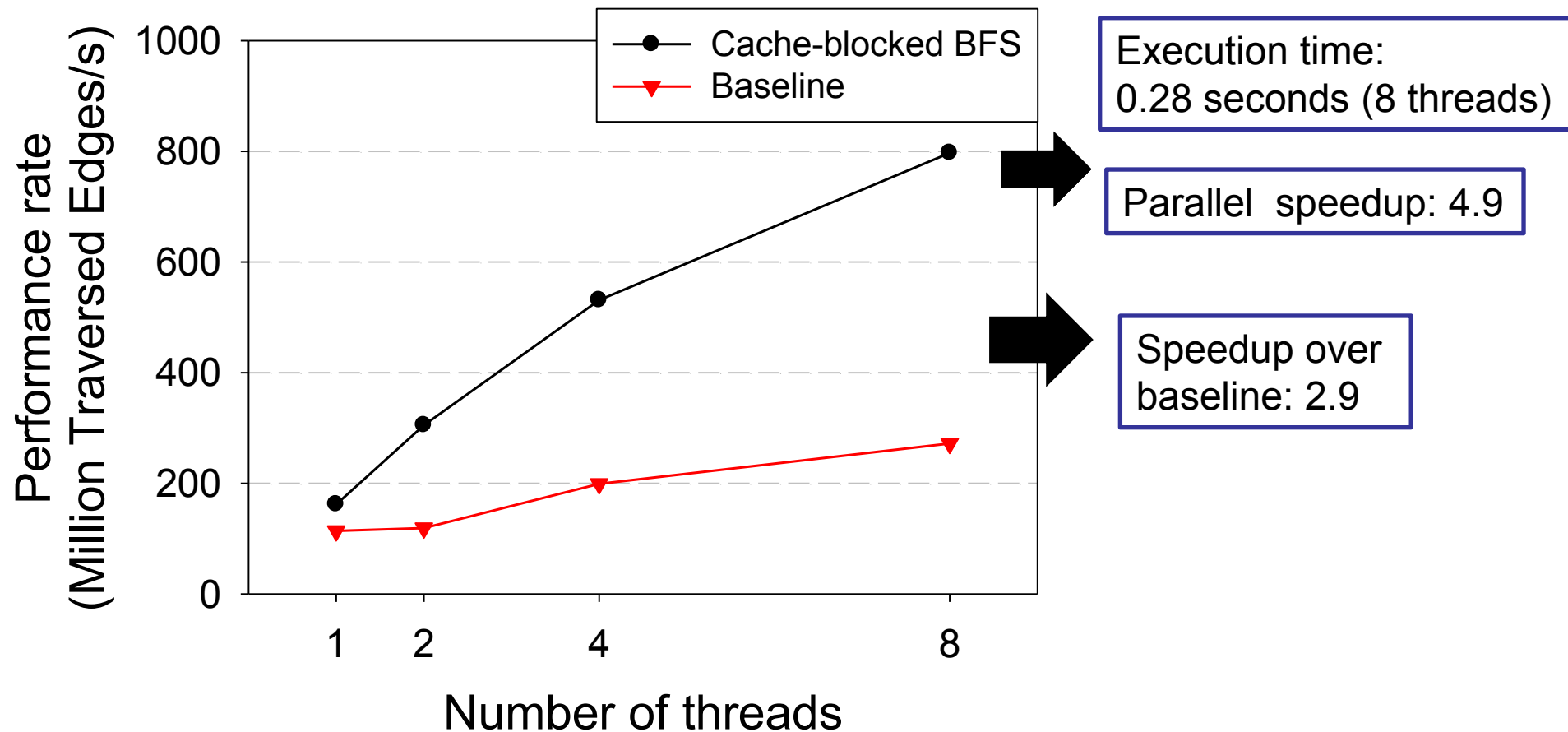
Cache locality improvement

Performance count: # of non-contiguous memory accesses (assuming cache line size of 16 words)
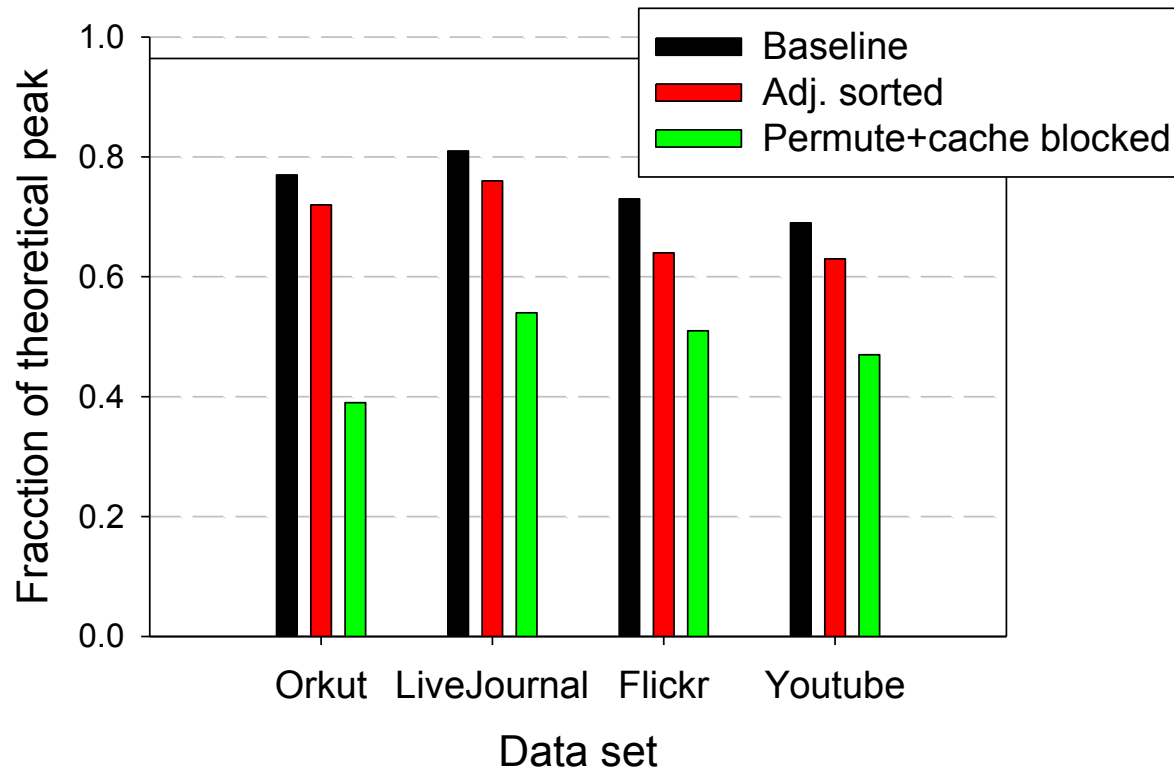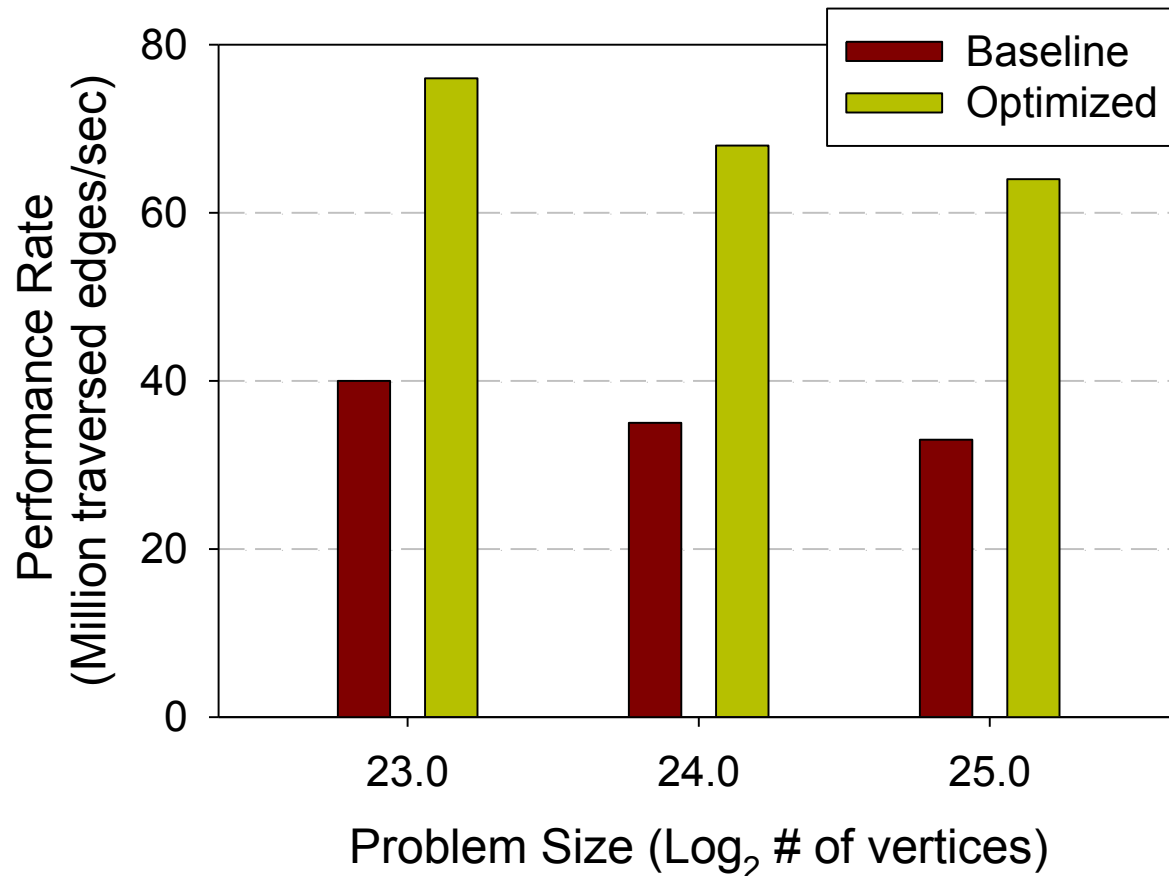

Theoretical count of the number of non-contiguous memory accesses: m+3n

Parallel performance (Orkut graph)

Graph: 3.07 million vertices, 220 million edges
Single socket of Intel Xeon 5560 (Core i7)

Performance Improvement (Betweenness Centrality on synthetic R-MAT networks)

Parallel performance on 8 threads of Xeon 5560.

Fast atomics in baseline. New optimizations:
- Sorted adjacencies
- Hugepages
- Software Prefetching
- Aligned mem. accesses

Conclusions

- New cache-blocking formulation to enhance cache locality and performance of parallel BFS.

- Small-world networks can be preprocessed to significantly reduce the number of non-contiguous memory accesses.

- Up to a 3x performance improvement over previous optimized parallel implementation.

Future Work

- Graph compression to reduce memory footprint.

- Extending the cache blocking formulation to more complex graph problems based on BFS.

- Parallelization strategies and optimizing communication on distributed memory systems.
  - avoid p-way graph partitioning

Thank you!

- Questions?

Kamesh Madduri

KMadduri@lbl.gov

madduri.org

SNAP (Small-world Network Analysis and Partitioning) on Sourceforge

http://snap-graph.sourceforge.net/