Irregular Graph Algorithms on Modern Multicore, Manycore, and Distributed Processing Systems

Doctoral Defense

George M. Slota

Scalable Computing Laboratory
Department of Computer Science and Engineering
The Pennsylvania State University

22 Feb. 2016

Defense Overview

- Motivation
- Summaries of Research
- Manycore graph processing
- Distributed graph processing
- Conclusions

Motivation

- Graph analysis is key for the study of biological, chemical, social, and other networks
- Real-world graphs are big, irregular, complex
  - Graph analytics is one of DARPA's 23 toughest mathematical challenges
  - Facebook graph: 1.6B people, 500B friendships
  - Brain graph: 100B neurons, 1,000T synaptic connections
  - Skewed degree distributions, small-world nature make parallelization difficult
- Modern computational systems are also big and complex
  - Multiple levels of parallelism, memory hierarchy, configurations
  - Heterogeneous – host, GPU, coprocessors (Xeon Phi)
  - Optimization – account for socket-level, node-level, and distributed

Motivation
Goals of Research

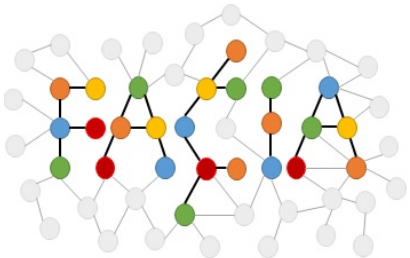
- How do we design parallel graph algorithms for computational efficiency under all of the aforementioned difficulties?
- What algorithmic traits are common to various irregular graph algorithms that we can optimize for?
- How do we store/organize and access graphs and associated data efficiently in shared and distributed memory?

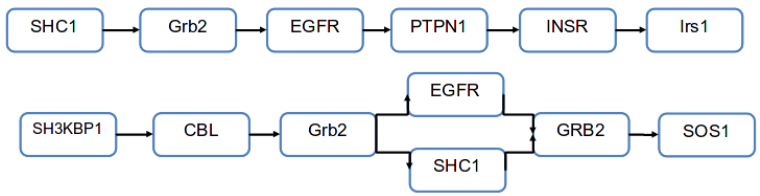
Research Summaries

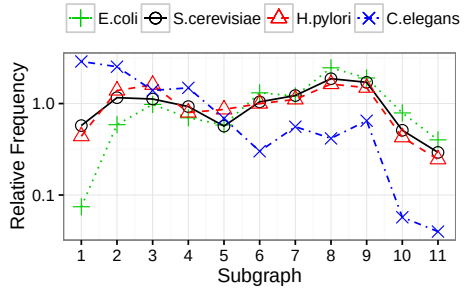
Summaries of Research: Color-coding, Connectivity, Partitioning, In-memory Layout

Topic 1: Summary of Color-coding

FASCIA subgraph enumeration and FASTPATH min-weight path finding







Overview:

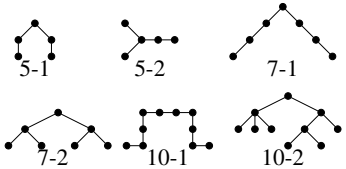
- Subgraph Counting: Find and count the number of occurrences of some input template in a larger graph (∼NP-complete)
- Minimum Weight Path Finding: In an edge-weighted network, find the k length simple path with the least sum of weights (NP-hard)
- Applications: Motif/anti-motif finding, network classification/clustering, network alignment, signaling pathways detection

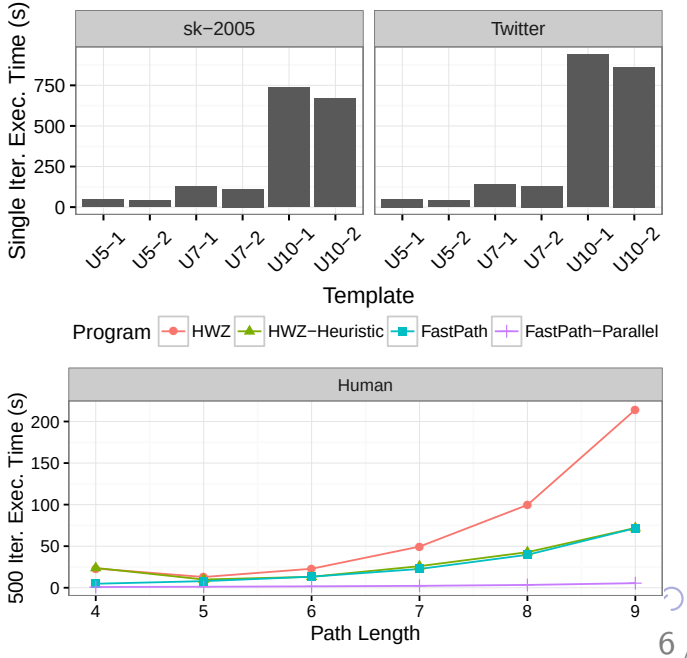
Contributions:

- Highly optimized implementation of the color-coding technique [Alon et al., 2008] for tree-structured subgraph counting; up to five orders-of-magnitude faster than prior work
- Extended baseline code for minimum weight path finding

Software:

- **Fascia**: Fast Approximate Subgraph Counting
- **FastPath**: Fast Minimum Weight Path Finding





Topic 2: Summary of Graph Connectivity Research

Multistep and BiCC Algorithms

Overview:

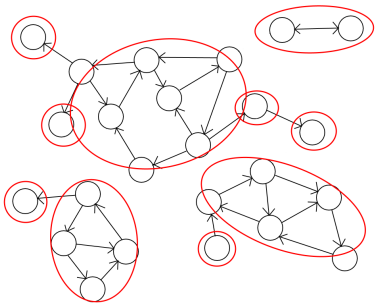
- Connectivity and Weak Connectivity: Find maximal components of a graph where a path exists between all vertices (ignore directivity of edges for weak)
- Strong Connectivity: Find maximal components of a directed graph where each vertex in a component has a path to every other vertex in the component
- Biconnectivity: Find maximal components of an undirected graph where the removal of any single vertex would not disconnect the component
- Applications: Social network analysis, scientific computing, network resilience analysis

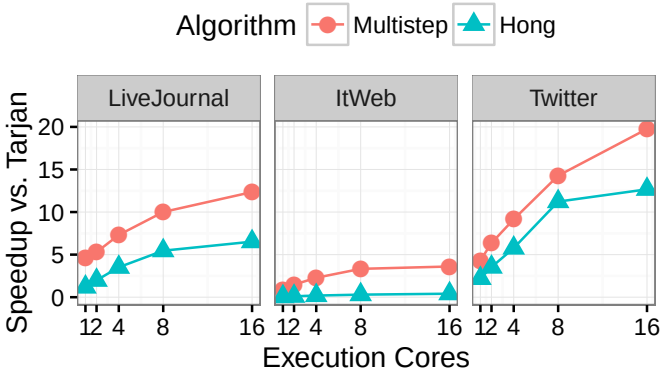
Contributions:

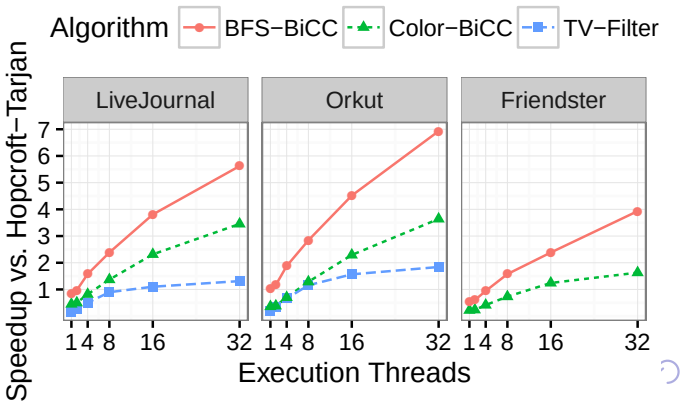
- New parallel algorithms for connected (CC), weakly connected (WCC), strongly connected (SCC), and biconnected components (BiCC)
- Identified several shared-memory optimizations (multi-level queues, minimize synchronizations, efficient algorithm design)
- Demonstrated over **2-7x average speedup** to state-of-the-art

Software:

- **Multistep**: Parallel methods for CC, WCC, and SCC
- **BiCC**: Open-source implementations available

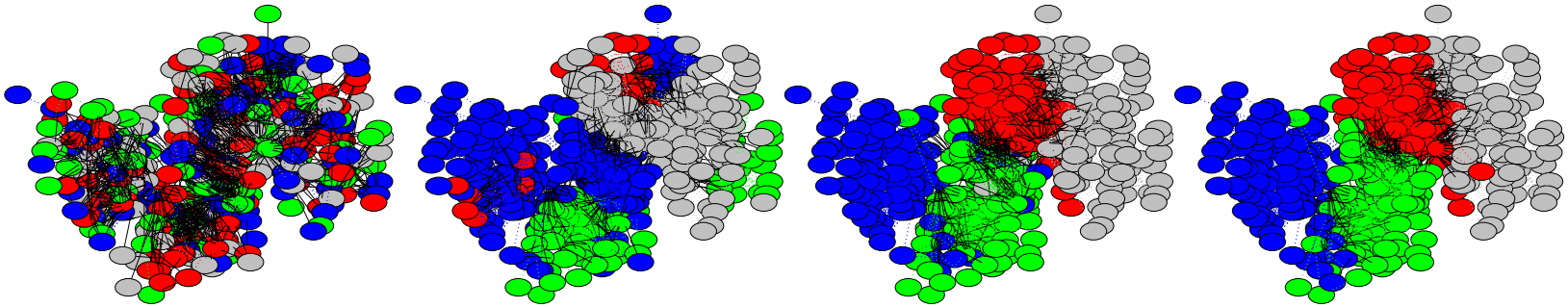






Topic 3: Summary of Graph Partitioning Research

PuLP: Partitioning Using Label Propagation



Overview:

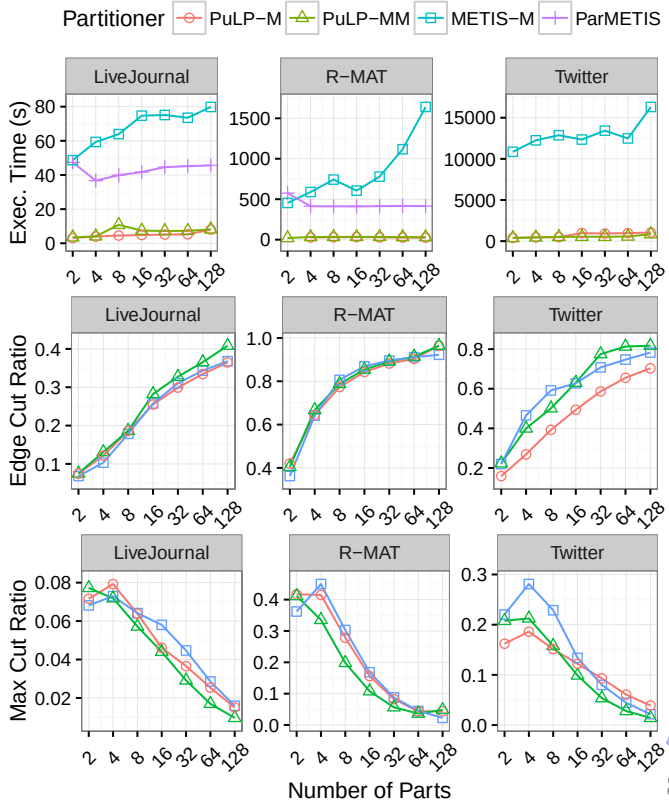
- Graph Partitioning: Separate a graph into k balanced *parts* with minimal inter-part edges (edge cut) for distributed computations (NP-complete problem)
- Label Propagation: Efficient (linear work) and scalable (naïvelyparallel) community detection and clustering algorithm [Raghavan et al., 2007]

Contributions:

- Parallel (OpenMP and soon MPI) multi-constraint multi-objective graph partitioning method for small-world networks that exploits Label Propagation community detection algorithm
- On suite of test graphs, **14.5× faster** and **38× less memory** on average relative to (Par)METIS with comparable or better cut quality
- Distributed version scales to **hundred billion edge networks**

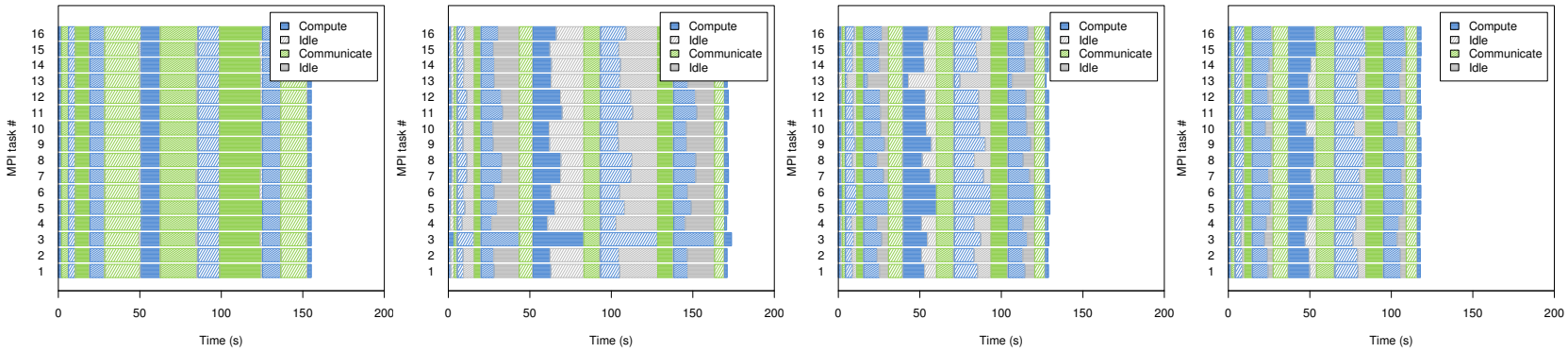
Software:

- **PuLP**: Interface now available in **Zoltan2** package of **Trilinos**



Topic 4: Summary of Distributed Layout Research

DGL: Distributed Graph Layout



Overview:

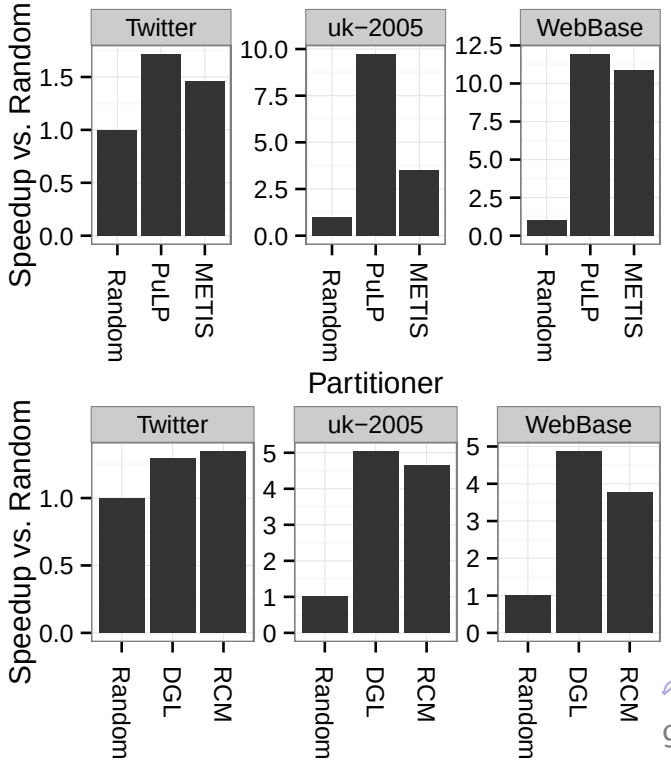
- Graph Layout: How to store a graph in distributed memory in terms of graph partitioning and intra-node vertex ordering

Contributions:

- Methodology for distributed in-memory layout for graphs
- Uses PuLP for partitioning and novel BFS-based ordering scheme
- Speedups for distributed computation (**up to 4x**) and communication (**up to 12x**) relative to naive methods for running PageRank with PuLP and DGL ordering
- Both PuLP and DGL ordering are faster to compute than other methods

Software:

- **DGL**: Distributed Graph Layout with PuLP and DGL ordering



Graph Algorithms for Manycore

Part 1: **Manycore graph algorithms** – abstraction for wide parallelism

Motivating questions for this work

- **Q**: What are some **common abstractions** that we can use to develop parallel graph algorithms for manycores?
- **Q**: What **key optimization strategies** can we identify to design new parallel graph algorithms for manycores?
- **Q**: Is it possible to develop **performance-portable implementations** of graph algorithms using advanced **libraries and frameworks** using the above optimizations and abstractions?

Contributions

- **Q**: **Common abstractions** for manycores?
  - We use array-based data structures, express computation in the form of nested loops.
- **Q**: **Key optimization strategies**
  - We improve load balance by manual loop collapse, coalesce memory access, and use collective operations when possible.
- **Q**: **Performance-portable implementations** of graph algorithms using advanced **libraries and frameworks**?
  - We use Kokkos [Carter Edwards et al., 2014].
- We compare high-level implementations using new framework to hand-optimized code + vary graph computations + vary graph inputs + vary manycore platform.

Background
GPU and Xeon Phi microarchitecture

- **GPU**
  - Multiprocessors (up to about 15/GPU)
  - Multiple groups of stream processors per MP (12×16)
  - *Warps* of threads all execute SIMT on single group of stream processors (32 threads/warp, two cycles per instruction)
  - Irregular computation (high degree verts, if/else, etc.) can result in most threads in warp doing NOOPs
- **Xeon Phi (MIC)**
  - Many simple (Pentium 4) cores, 57-61
  - 4 threads per core, need at least 2 threads/core for OPs on each cycle
  - Highly vectorized (512 bit width) - difficult for irregular computations to exploit

Background
Kokkos and GPU microarchitecture

- Kokkos
  - Developed as back-end for portable scientific computing
  - Polymorphic multi-dimensional arrays for varying access patterns
  - Thread parallel execution for fine-grained parallelism
- Kokkos model - performance portable programming to multi/manycores
  - Thread team - multiple warps on same multiprocessor, but all still SIMT for GPU
  - Thread league - multiple thread teams, over all teams all work is performed
  - Work statically partitioned to teams before parallel code is called

- **Observation**: many (synchronous) graph algorithms follow a tri-nested loop structure
  - Optimize for this general algorithmic template
  - Transform structure for more parallelism

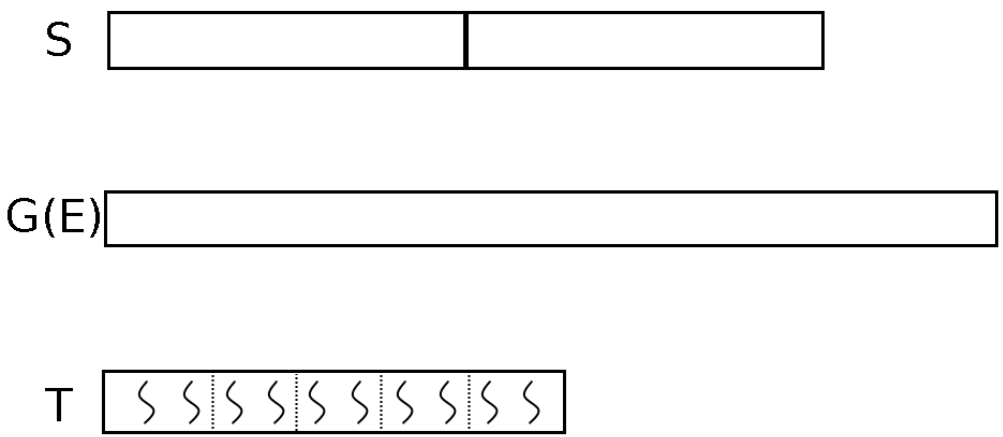
```
1:  Initialize temp/result arrays A_t[1..n], 1 ≤ t ≤ l.        ▷ l = O(1)
2:  Initialize S_1[1..n].
3:  for i = 1 to niter do                                      ▷ niter = O(log n)
4:      Initialize S_{i+1}[1..n].                              ▷ Σ_i |S_i| = O(m)
5:      for j = 1 to |S_i| do                                  ▷ |S_i| = O(n)
6:          u ← S_i[j]
7:          Read/update A_t[u], 1 ≤ t ≤ l.
8:          for k = 1 to |E[u]| do                             ▷ |E[u]| = O(n)
9:              v ← E[u][k]
10:             Read/update A_t[v].
11:             Read/update S_{i+1}.
12:         Read/update A_t[u].
```

- **Observation**: many (synchronous) graph algorithms follow a tri-nested loop structure
  - Optimize for this general algorithmic template
  - Transform structure for more parallelism

```
1:  Initialize temp/result arrays A_t[1..n], 1 ≤ t ≤ l.        ▷ l = O(1)
2:  Initialize S_1[1..n].
3:  for i = 1 to niter do                                      ▷ niter = O(log n)
4:      Initialize S_{i+1}[1..n].                              ▷ Σ_i |S_i| = O(m)
5:      for j = 1 to |S_i| do                                  ▷ |S_i| = O(n)
6:          u ← S_i[j]
7:          Read/update A_t[u], 1 ≤ t ≤ l.
8:          for k = 1 to |E[u]| do                             ▷ |E[u]| = O(n)
9:              v ← E[u][k]
10:             Read/update A_t[v].
11:             Read/update S_{i+1}.
12:         Read/update A_t[u].
```

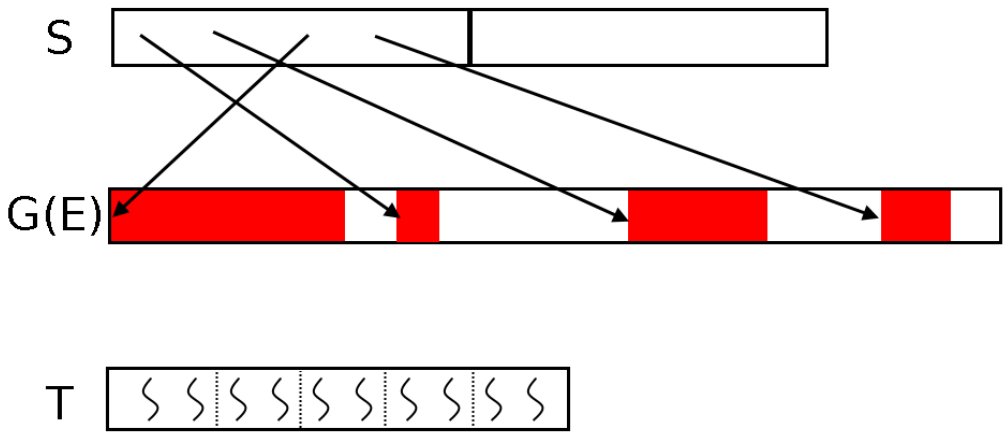
Graph Algorithms for Manycore
Parallelization strategies

- Baseline parallelization



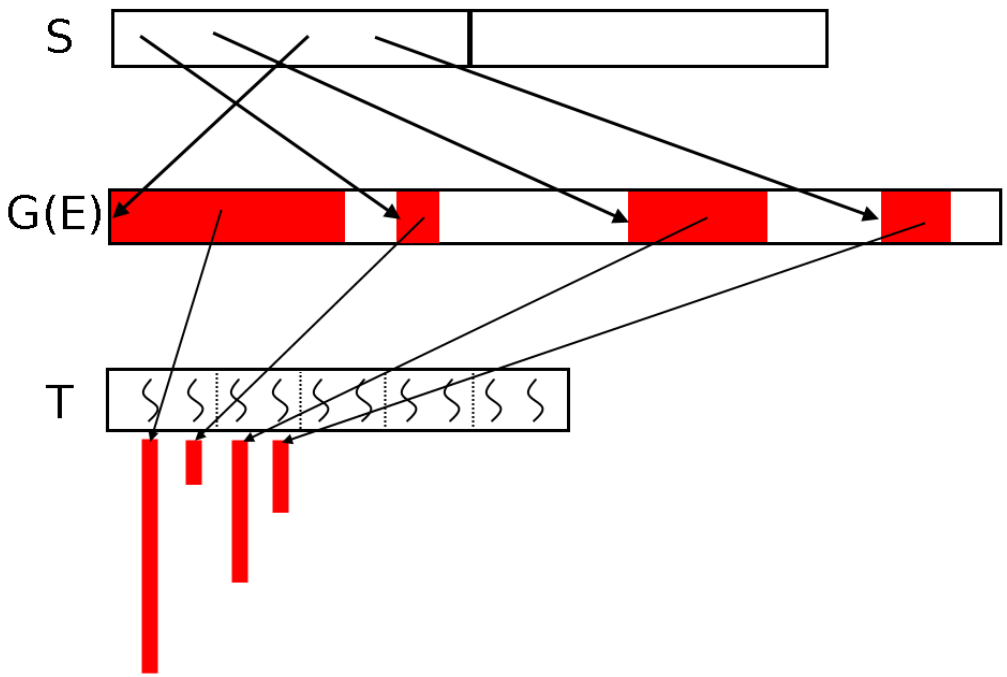
Graph Algorithms for Manycore
Parallelization strategies

- Baseline parallelization



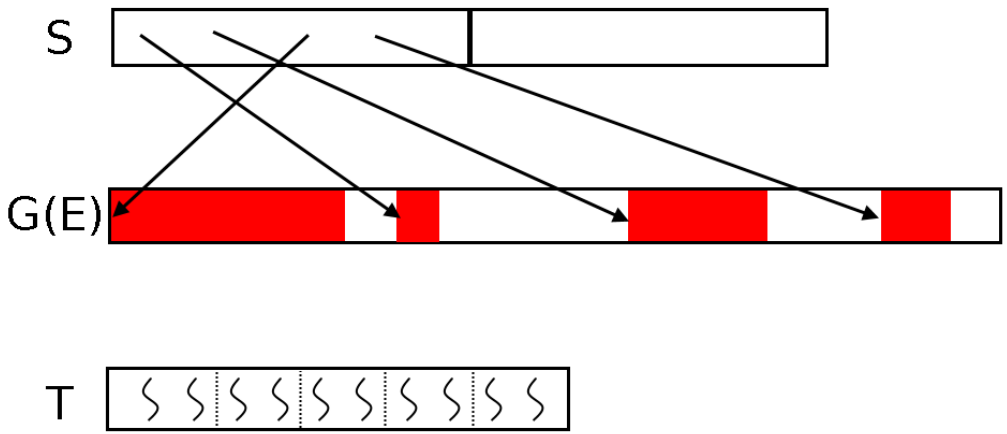
Graph Algorithms for Manycore
Parallelization strategies

- Baseline parallelization



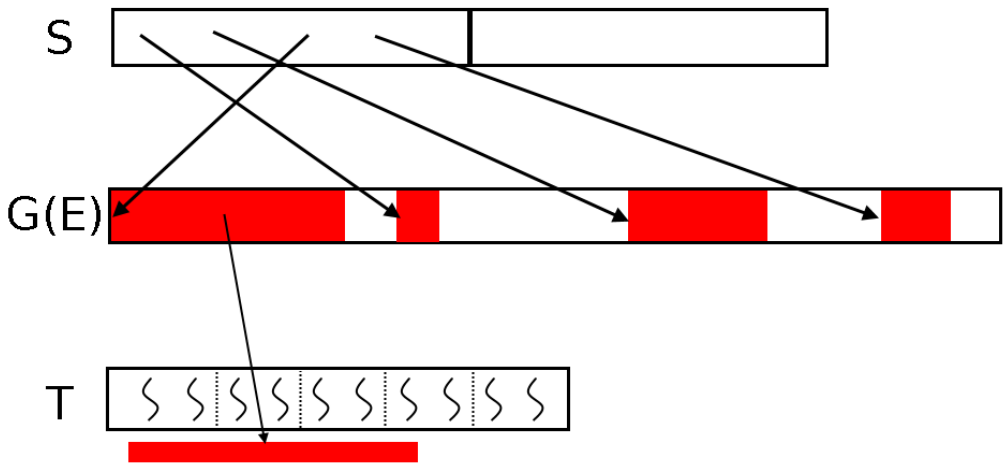
Graph Algorithms for Manycore
Parallelization strategies

- Baseline parallelization
- Hierarchical expansion (e.g. [Hong et al., 2011])



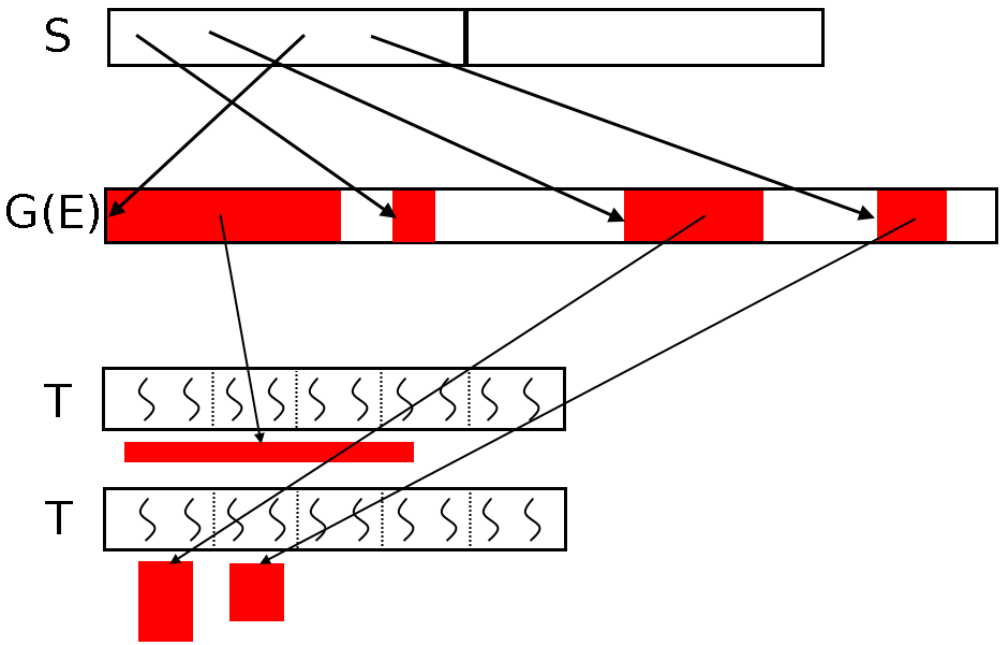
Graph Algorithms for Manycore
Parallelization strategies

- Baseline parallelization
- Hierarchical expansion (e.g. [Hong et al., 2011])



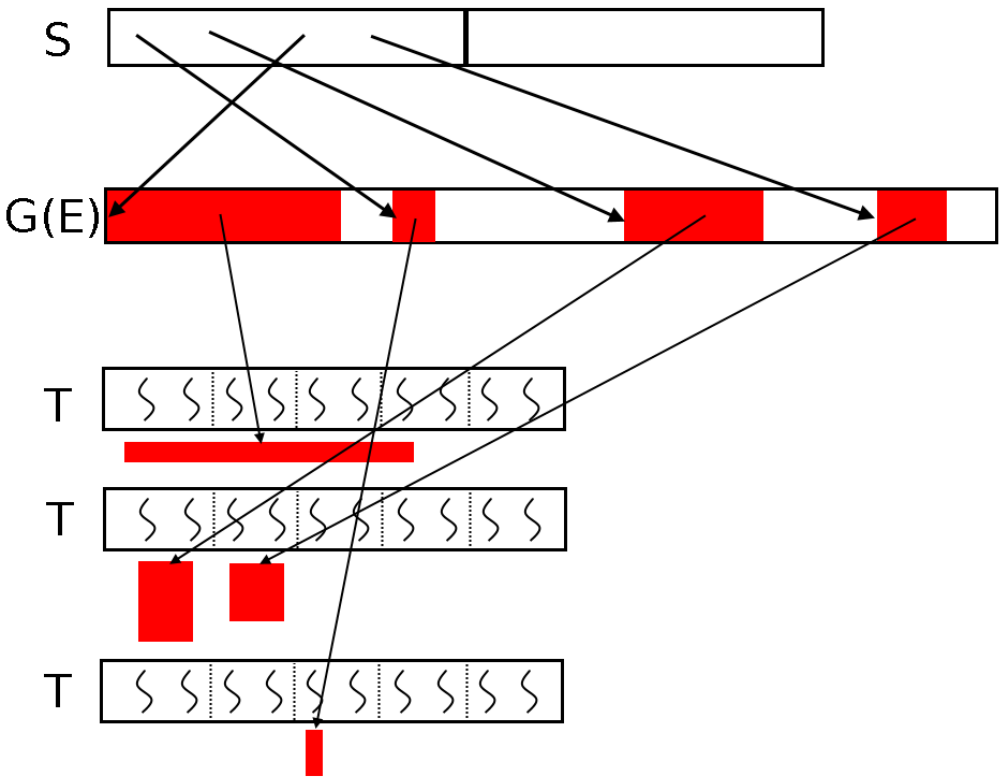
Graph Algorithms for Manycore
Parallelization strategies

- Baseline parallelization
- Hierarchical expansion (e.g. [Hong et al., 2011])



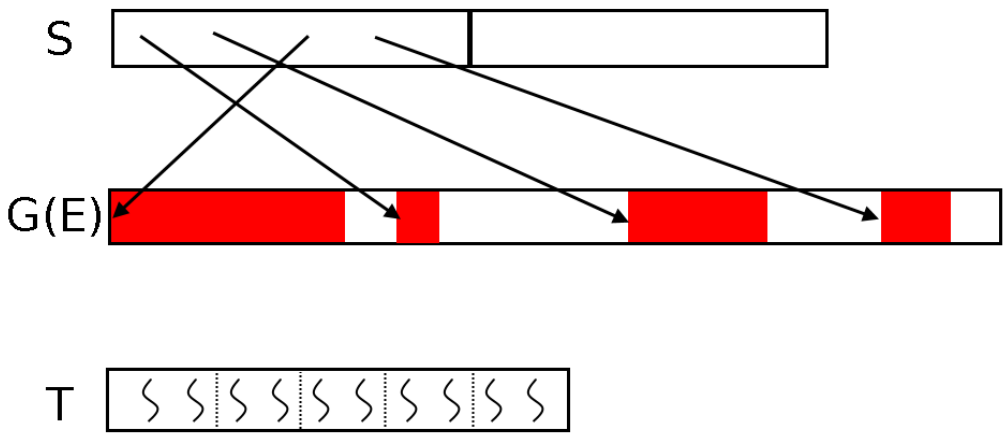
Graph Algorithms for Manycore
Parallelization strategies

- Baseline parallelization
- Hierarchical expansion (e.g. [Hong et al., 2011])



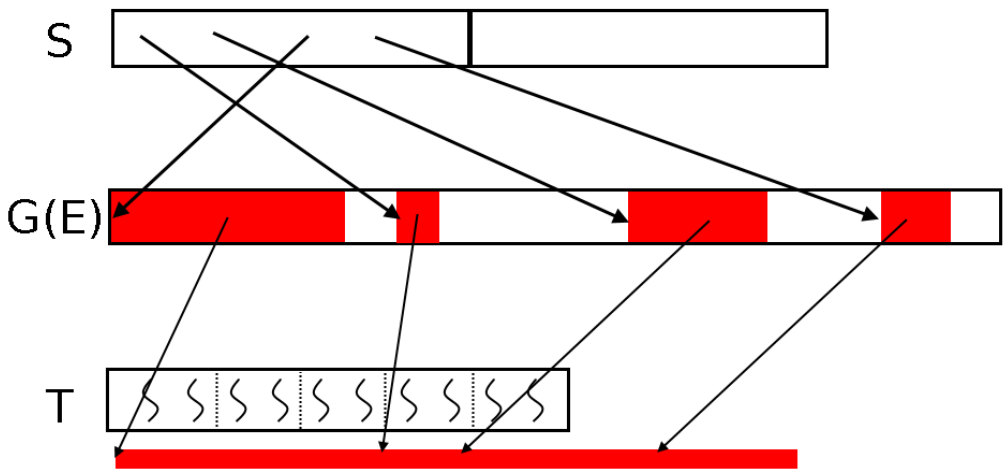
Graph Algorithms for Manycore
Parallelization strategies

- Baseline parallelization
- Hierarchical expansion (e.g. [Hong et al., 2011])
- 'Manhattan collapse - local' (e.g. [Merrill et al., 2012])



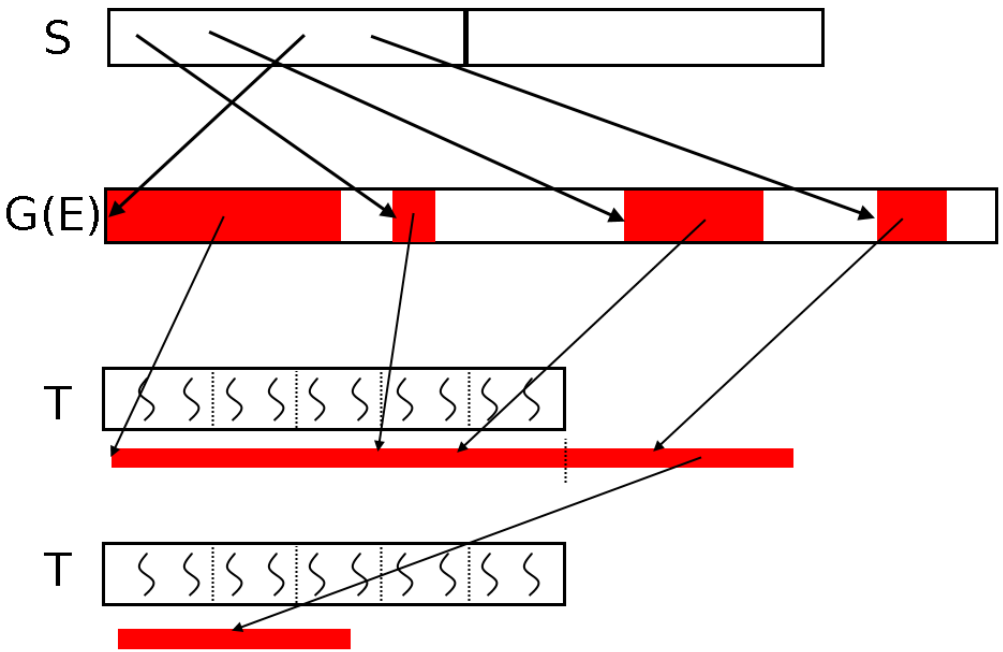
Graph Algorithms for Manycore
Parallelization strategies

- Baseline parallelization
- Hierarchical expansion (e.g. [Hong et al., 2011])
- 'Manhattan collapse - local' (e.g. [Merrill et al., 2012])



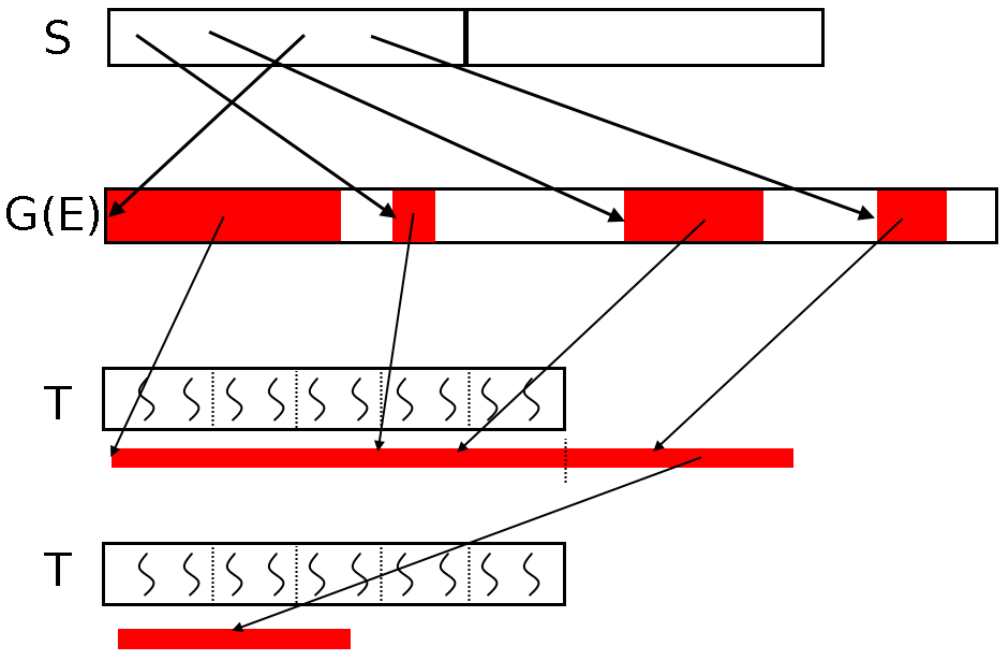
Graph Algorithms for Manycore
Parallelization strategies

- Baseline parallelization
- Hierarchical expansion (e.g. [Hong et al., 2011])
- 'Manhattan collapse - local' (e.g. [Merrill et al., 2012])



Graph Algorithms for Manycore
Parallelization strategies

- Baseline parallelization
- Hierarchical expansion (e.g. [Hong et al., 2011])
- 'Manhattan collapse - local' (e.g. [Merrill et al., 2012])
- 'Manhattan collapse - global' (e.g. [Davidson et al., 2014])



- Implementation
  - Develop with Kokkos for cross-platform compatibility
  - Implement MULTISTEP SCC algorithm for testing
- Memory access optimizations
  - Explicit shared memory utilization on GPU
  - Coalescing memory access (locality)
  - Minimize access to global/higher-level memory
- Collective operation optimizations
  - Warp and team-based operations (team scan, team reduce)
  - Minimize global atomics (team-based atomics)

Graph computations
Implemented algorithms

- Breadth-first search
- Color propagation
- Trimming
- The MULTISTEP algorithm [Slota et al., 2014] for Strongly Connected Components (SCC) decomposition

Graph computations
Breadth-first search

- Useful subroutine in other graph computations

```
1:  A_1[1..n] ← −1
2:  S_1[1] ← root
3:  level ← 0
4:  while |S_i| ≠ ∅ do
5:      Initialize S_{i+1}
6:      for j = 1 to |S_i| do
7:          u ← S_i[j]
8:          for k = 1 to |E[u]| do
9:              v ← E[u][k]
10:             if A_1[v] < 0 then
11:                 A_1[v] ← level
12:                 S_{i+1} ← v
13:     level ← level + 1
```

Graph computations
Breadth-first search

- Useful subroutine in other graph computations

```
1:  A_1[1..n] ← −1
2:  S_1[1] ← root
3:  level ← 0
4:  while |S_i| ≠ ∅ do
5:      Initialize S_{i+1}
6:      for j = 1 to |S_i| do
7:          u ← S_i[j]
8:          for k = 1 to |E[u]| do
9:              v ← E[u][k]
10:             if A_1[v] < 0 then
11:                 A_1[v] ← level
12:                 S_{i+1} ← v
13:     level ← level + 1
```

Graph computations
Color propagation

- Basic algorithm for connectivity
- General approach applies to other algorithms (e.g., label propagation)

```
1:  A_1[1..n] ← [1..n]
2:  S_1[1..n] ← [1..n]
3:  while |S_i| ≠ ∅ do
4:      Initialize S_{i+1}
5:      for j = 1 to |S_i| do
6:          u ← S_i[j]
7:          for k = 1 to |E[u]| do
8:              v ← E[u][k]
9:              if A_1[u] > A_1[v] then
10:                 A_1[v] ← A_1[u]
11:                 S_{i+1} ← v
```

Graph computations
Color propagation

- Basic algorithm for connectivity
- General approach applies to other algorithms (e.g., label propagation)

```
1:  A_1[1..n] ← [1..n]
2:  S_1[1..n] ← [1..n]
3:  while |S_i| ≠ ∅ do
4:      Initialize S_{i+1}
5:      for j = 1 to |S_i| do
6:          u ← S_i[j]
7:          for k = 1 to |E[u]| do
8:              v ← E[u][k]
9:              if A_1[u] > A_1[v] then
10:                 A_1[v] ← A_1[u]
11:                 S_{i+1} ← v
```

Graph computations
Trimming

- Routine for accelerating connectivity decomposition
- Iteratively trim 0-degree vertices

```
1:  A_1[1..n] ← 1
2:  S_1[1..n] ← [1..n]
3:  while |S_i| ≠ ∅ do
4:      Initialize S_{i+1}
5:      for j = 1 to |S_i| do
6:          u ← S_i[j]
7:          trim ← true
8:          for k = 1 to |E[u]| do
9:              v ← E[u][k]
10:             if A_1[v] = 1 then
11:                 trim ← false
12:         if trim = true then
13:             A_1[u] ← 0
14:             S_{i+1} ← E[u]
```

Graph computations
Trimming

- Routine for accelerating connectivity decomposition
- Iteratively trim 0-degree vertices

```
1:  A_1[1..n] ← 1
2:  S_1[1..n] ← [1..n]
3:  while |S_i| ≠ ∅ do
4:      Initialize S_{i+1}
5:      for j = 1 to |S_i| do
6:          u ← S_i[j]
7:          trim ← true
8:          for k = 1 to |E[u]| do
9:              v ← E[u][k]
10:             if A_1[v] = 1 then
11:                 trim ← false
12:         if trim = true then
13:             A_1[u] ← 0
14:             S_{i+1} ← E[u]
```

Graph computations

Multistep SCC decomposition [Slota et al., 2014]

- Combination of trimming, BFS, and color propagation

1: $T \leftarrow \text{Trim}(G)$
2: $V \leftarrow V \setminus T$
3: Select $v \in V$ for which $d_{in}(v) * d_{out}(v)$ is maximal
4: $D \leftarrow \text{BFS}(G(V, E(V)), v)$
5: $S \leftarrow D \cap \text{BFS}(G(D, E'(D)), v)$
6: $V \leftarrow V \setminus S$
7: **while** $\text{NumVerts}(V) > 0$ **do**
8: $\quad C \leftarrow \text{ColorProp}(G(V, E(V)))$
9: $\quad V \leftarrow V \setminus C$

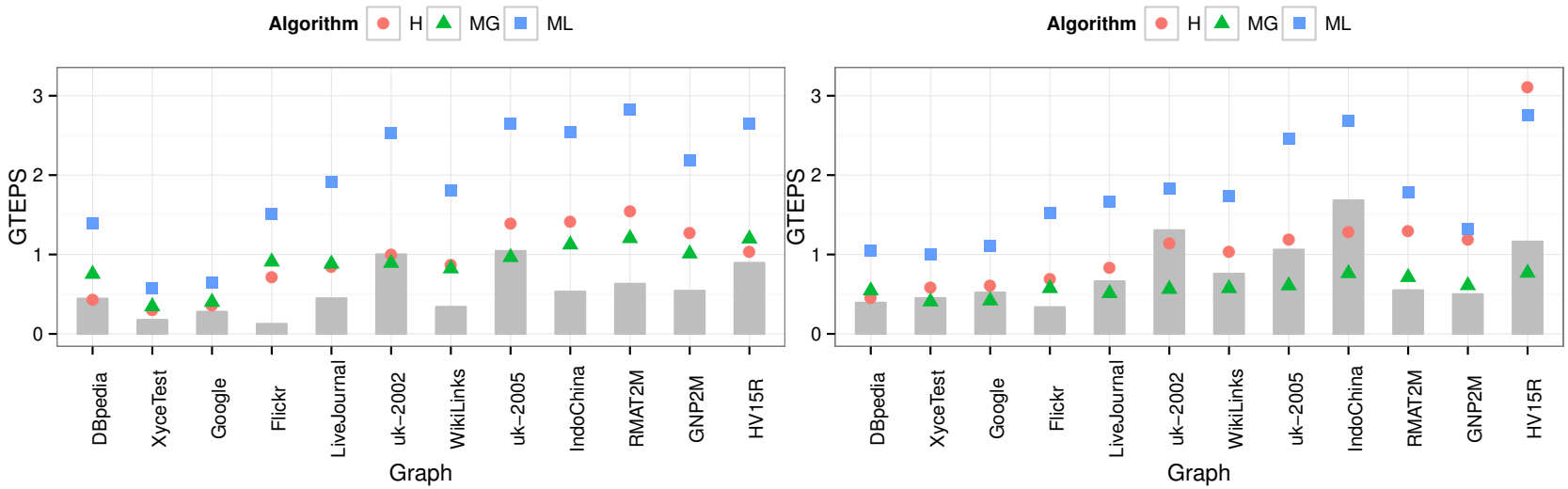
Performance Results

Experimental Setup

- Test systems: One node of *Shannon* and *Compton* at Sandia and *Blue Waters* at the NCSA
  - Intel Xeon E5-2670 (Sandy Bridge), dual-socket, 16 cores, 64-128 GB memory
  - NVIDIA Tesla K40M GPU, 2880 cores, 12 GB memory
  - NVIDIA Tesla K20X GPU, 2688 cores, 6 GB memory
  - Intel Xeon Phi (KNC, ∼3120A), 228 cores, 6 GB memory
- Test graphs:
  - Various real and synthetic small-world graphs, 5.1 M to 936 M edges
  - Social networks, circuit, mesh, RDF graph, web crawls, R-MAT and $G(n, p)$ random graphs, Wikipedia article links

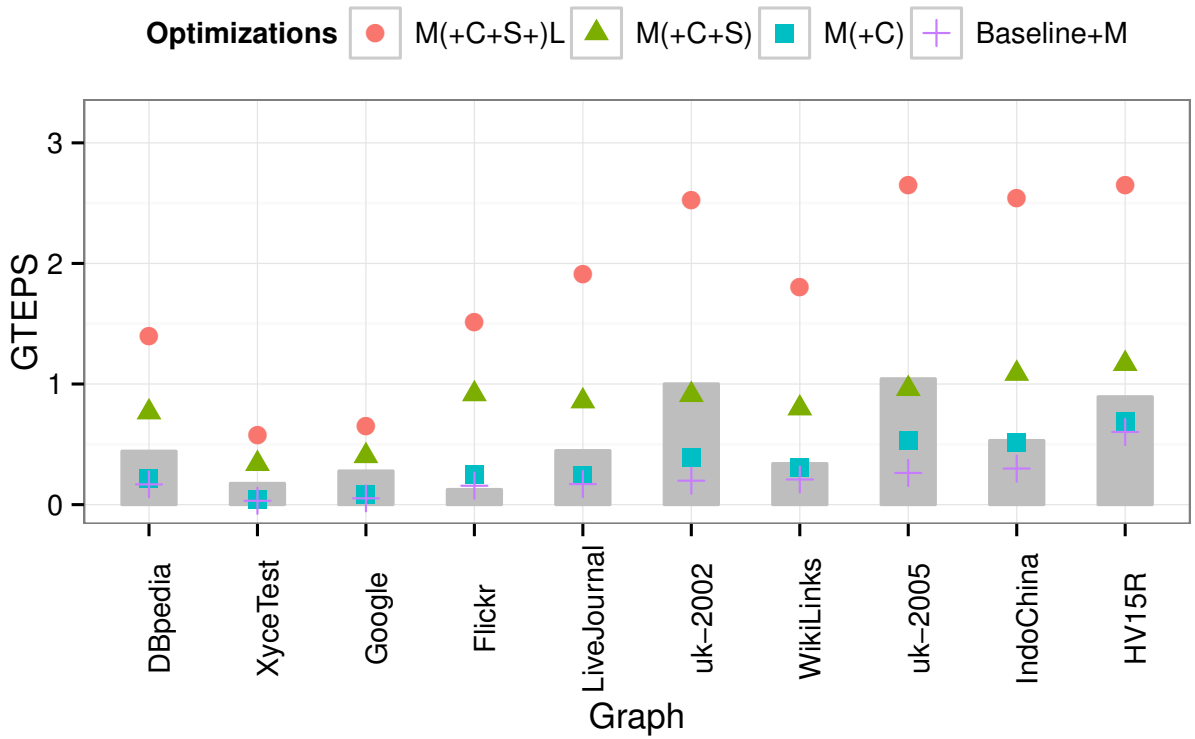
BFS and Coloring versus Loop Strategies
Tesla K40M

- Performance in GTEPS (10^9 trav. edges per second) for BFS (left) and color propagation (right) on Tesla K40M.
- Graphs ordered by increasing density from left to right
- Gray Bar: Baseline, H: Hierarchical, ML: Local collapse, MG: Global collapse,



BFS - Cumulative Impact of Optimizations
Tesla K40M

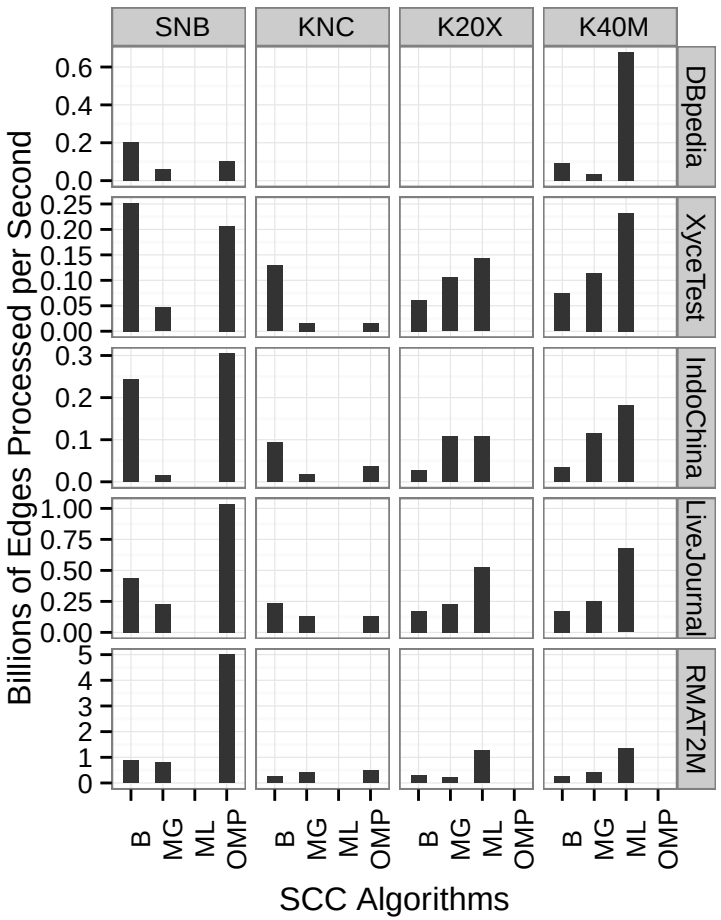
- Gray Bar: baseline, M: local collapse, C: coalescing memory access, S: shared memory use, L: local team-based primitives



SCC Cross-platform Performance Comparison

Sandy bridge (SB), MIC Knight's Corner (KNC), K40M, K20M

- B: Baseline, MG: Manhattan Global, ML: Manhattan Local, OMP: Optimized OpenMP code



Conclusions

- We express several graph computations in the **Kokkos** programming model using an **algorithm design abstraction** that allows portability across both multicore platforms and **accelerators**.
- The **SCC** code on GPUs (using the **Local Manhattan Collapse** strategy) demonstrates up to a 3.25× speedup relative to a state-of-the-art parallel CPU implementation running on a dual-socket compute node.
- Future work: Expressing other computations using this framework; Heterogeneous CPU-GPU processing; Newer architectures.

Large Graph Analysis – Full-scale Optimization

Part 2: **Large Graph Analysis** – Techniques for multi-billion vertex scale graph analysis

Large Graph Analysis – Full-scale Optimization

Current state-of-the-art:

- Many frameworks exist for large graph analysis
- Single-node parallel frameworks demonstrate good performance, but are limited by shared-memory (Ligra, Green-Marl, etc.)
- Disk-based frameworks are slow at all scales (MapReduce-like)
- Most distributed-memory frameworks often run slower than serial C code without massive parallelism, also fail for large graphs (GraphX, GraphLab and its variants, most others)
- Some can scale to large graphs, but only give reasonable relative performance at that scale (Giraph)
- Some are performant, but require specialized hardware (FlashGraph)

Motivating questions for this work

- **Q**: What is needed to analyze the **largest publicly-available graph instances** on modern distributed **HPC** systems ?
- **Q**: What **optimizations strategies and abstractions** can we identify to simplify implementation efforts ?
- **Q**: Is it possible to be both **highly performant** while keeping the implementation effort as **simple** as possible ?

Contributions

- **Q**: **Needed to analyze large graphs on HPC** ?
  - We develop optimized MPI+OpenMP code that uses 256 nodes of the Blue Waters supercomputer to analyze the 3.5B page and 130B link 2012 Web Data Commons web crawl.
- **Q**: **Optimizations strategies and abstractions** ?
  - We recognize three classes of communication patterns common to many graph algorithms and develop optimizations for them.
- **Q**: **Highly performant and simple** ?
  - Yes. Each algorithm can be implemented in only a couple hundred lines of C++ code while outperforming state-of-the-art frameworks by orders-of-magnitude.

Research Goals

Goals of this research:

- Develop methodology for analyzing the **largest publicly available graph instances** on **HPC systems**
- Identify **optimization strategies and abstractions** that can simplify implementation efforts
- Strive for **ease of implementation** while retaining **high performance**, identify and evaluate potential performance and scalability tradeoffs
- Use these implementations to gain novel insight into graph structure

Design Tradeoffs and Considerations

Tradeoffs (ease of implementation vs. scalability):

- **1D** (vertex-based) vs. 2D (edge-based) partitioning and graph layout
- **Bulk-synchronous** vs. asynchronous communication
- Programming language and parallel programming model
  - High-level language vs. **C/C++**
  - High-level model vs. MPI only vs. **MPI+OpenMP**

Other considerations:

- In-memory graph representation
  - Compressed vs. **uncompressed**, efficiency in accessing structural information
- Partitioning strategy (with 1D layout)
  - Vertex block vs. Edge block vs. Random
- Generalizability for many graph algorithms

Communication in Distributed Graph Algorithms

Observation: many iterative graph algorithms have similar communication patterns

- *BFS-like*: build and exchange global queue of vertices on each iteration
- *PageRank-like*: exchange all per-vertex values on each iteration
- *ColorProp-like*: exchange subset of per-vertex values on each iteration

Takeaway: develop optimized outlines for each of these patterns, fill in algorithm-specific details

BFS-like Algorithms

- Build and exchange global queue (Q) of vertices on each iteration

1: $Q \leftarrow v_{root}$
2: $Data[V_{local} \cup V_{ghost}] \leftarrow$ InitData()
3: **while** $Q \neq \varnothing$ **do**
4: **for all** $v \in Q$ **do**
5: $Data[v] \leftarrow$ PerformWork($v, Data$)
6: **for all** $\langle v, u \rangle \in E$ **do**
7: $Data[u] \leftarrow$ PerformWork($u, Data$)
8: **if** CriteraSatisfied() **then**
9: $Q_n \leftarrow u$
10: $Q \leftarrow$ Exchange(Q_n)

PageRank-like Algorithms

- Exchange all per-vertex *Data* with neighbors on each iteration

1: $Data[V_{local} \cup V_{ghost}] \leftarrow \text{InitData}()$
2: **for** $iter = 1 \cdots NumIter$ **do**
3: **for all** $v \in V_{local}$ **do**
4: $Data[v] \leftarrow \text{PerformWork}(v, Data)$
5: **for all** $\langle v, u \rangle \in E$ **do**
6: $Data[v] \leftarrow \text{PerformWork}(u, Data)$
7: $Q_{Data} \leftarrow Data[v]$
8: $Data[V_{ghost}] \leftarrow \text{Exchange}(Q_{Data})$

ColorProp-like Algorithms

- Exchange subset of per-vertex *Data* on each iteration

1: $Q \leftarrow V_{local}$
2: $Data[V_{local} \cup V_{ghost}] \leftarrow \text{InitData}()$
3: **for** $iter = 1 \cdots NumIter$ **or** $Q \neq \varnothing$ **do**
4: **for all** $v \in Q$ **do**
5: $Data[v] \leftarrow \text{PerformWork}(v, Data)$
6: **for all** $\langle v, u \rangle \in E$ **do**
7: $Data[v] \leftarrow \text{PerformWork}(u, Data)$
8: **if** CriteraSatisfied() **then**
9: $Q_n \leftarrow v$
10: $Q_{Data} \leftarrow Data[v]$
11: $Q \leftarrow \text{Exchange}(Q_n)$
12: $Data[V_{ghost}] \leftarrow \text{Exchange}(Q_{Data})$

Distributed Graph Processing

Large-scale graph analysis on HPC systems

- Optimized parallel graph I/O and pre-processing
- Fully parallel – almost no serial work (tasks & threads)
- Implemented algorithms:
  - *BFS-like*: MULTISTEP SCC (1st stage), MULTISTEP WCC (1st stage), K-core, Harmonic Centrality
  - *PageRank-like*: PageRank, Label Propagation (this implementation)
  - *ColorProp-like*: MULTISTEP WCC (2nd stage), PULP
- Compact and efficient: ~2,000 total lines of code

Performance Results

Experimental Setup

Test systems and test networks

- Test systems: and *Compton* at Sandia and *Blue Waters* at NCSA
  - Intel Xeon E5-2670, dual-socket, 16 cores, 64 GB memory
  - AMD Interlagos 6276, dual-socket, 16 cores, 64 GB memory

Graph	n	m	D_{avg}	Source
Web Crawl	3.5 B	129 B	36	Meusel et al. [2015]
R-MAT	3.5 B	129 B	36	Chakrabarti et al. [2004]
$G(n,p)$	3.5 B	129 B	36	
R-MAT	2^{25}-2^{32}	2^{29}-2^{36}	16	Chakrabarti et al. [2004]
$G(n,p)$	2^{25}-2^{32}	2^{29}-2^{36}	16	
Host	89 M	2.0 B	22	Meusel et al. [2015]
Pay	39 M	623 M	16	Meusel et al. [2015]
Twitter	53 M	2.0 B	38	Cha et al. [2010]
LiveJournal	4.8 M	69 M	14	Leskovec et al. [2009]
Google	875 K	5.1M	5.8	Leskovec et al. [2009]

End-to-end Analysis

256 nodes of Blue Waters

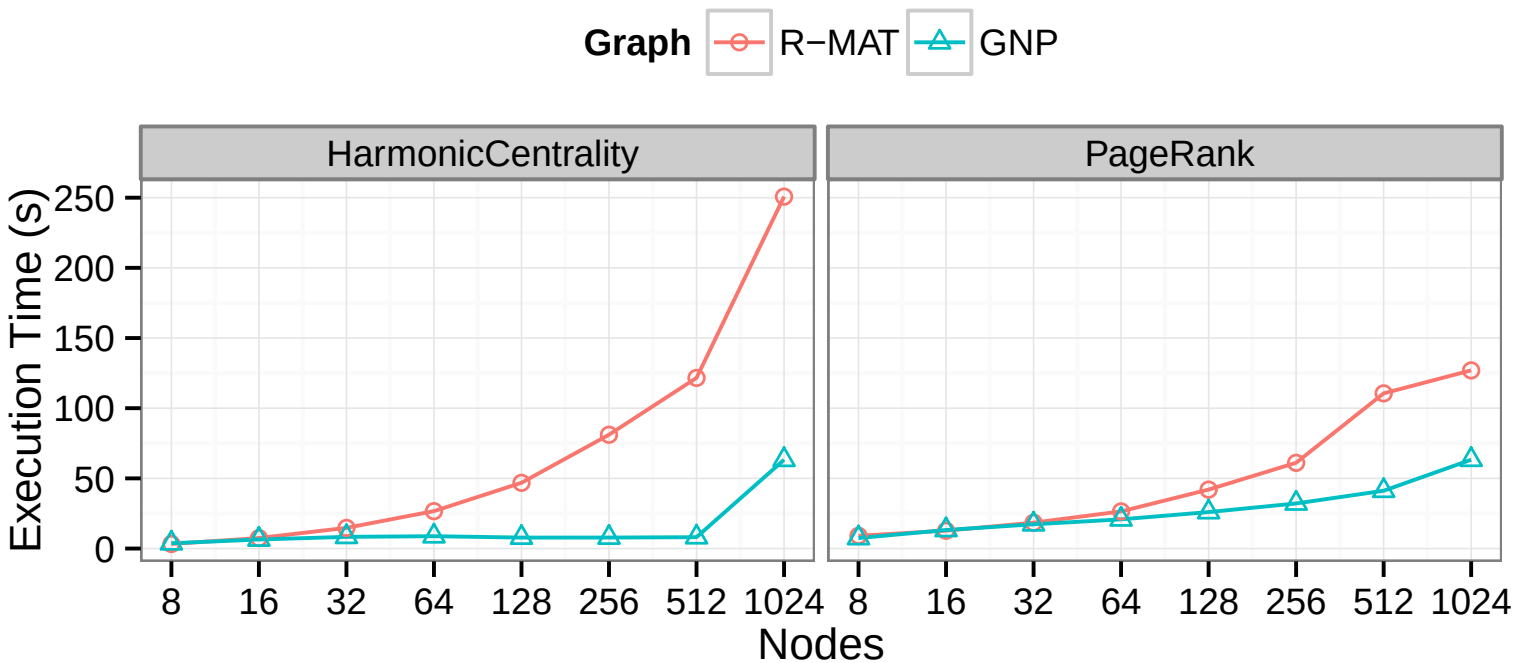
- Ran all six analytics on the web crawl (with three partitioning strategies) and the $G(n, p)$ and R-MAT graphs
- With $\frac{n}{p}$ partitioning strategy, end-to-end times are about 20 minutes (3 minutes for I/O+PP), expect up to 50% further reduction possible with PuLP

Analytic	WC-np	WC-mp	WC-rand	R-MAT	$G(n, p)$
PageRank	87	111	227	125	121
Label Propagation	400	435	367	993	992
WCC	88	63	112	68	77
Harmonic Centrality	54	46	101	252	84
K-core	445	363	583	579	481
SCC	184	108	184	89	83

Weak Scaling

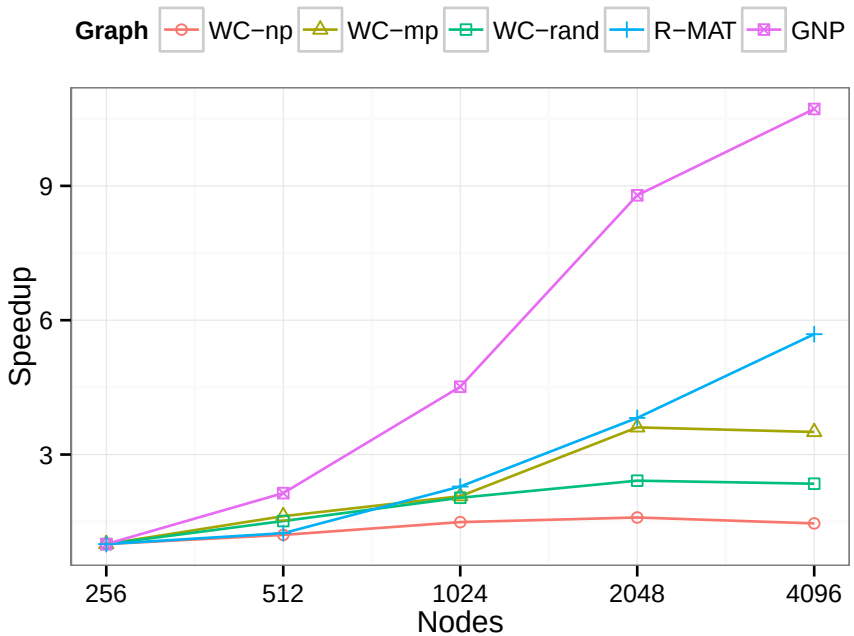
Blue Waters from 8 to 1024 nodes

- Harmonic Centrality and PageRank on R-MAT and $G(n, p)$ graphs with vertex block partitioning
- 2^{22} vertices per node and 2^{26} edges per node



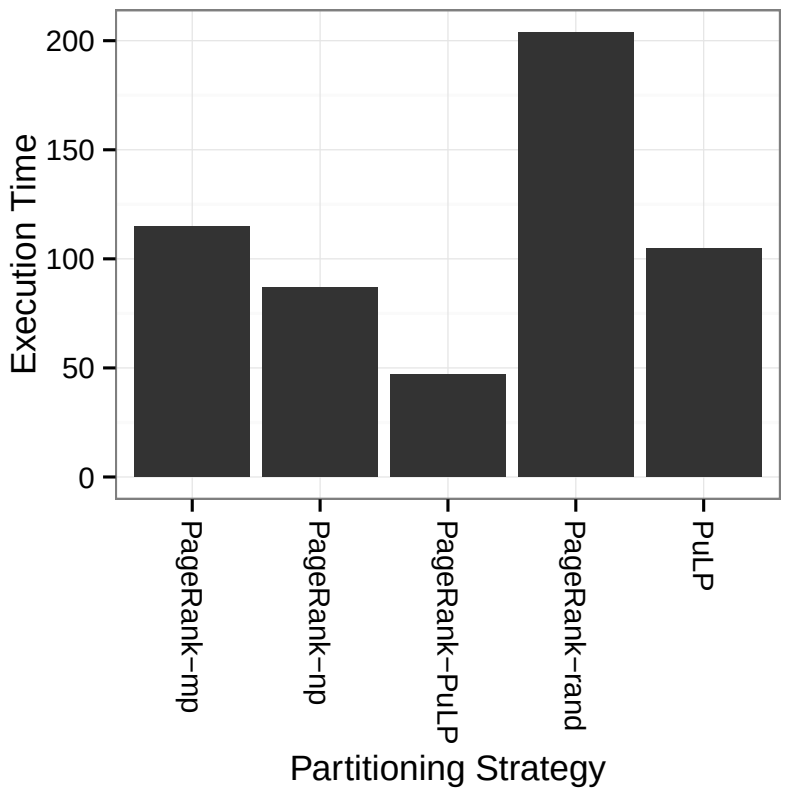
Strong Scaling
Blue Waters from 256 to 4096 nodes

- Label propagation – shows strong scaling from 256 to 4096 nodes
- PageRank-like and ColorProp-like strong scale nicely; BFS-like more dependent on graph structure (high number of synchronizations and low computation per iteration)



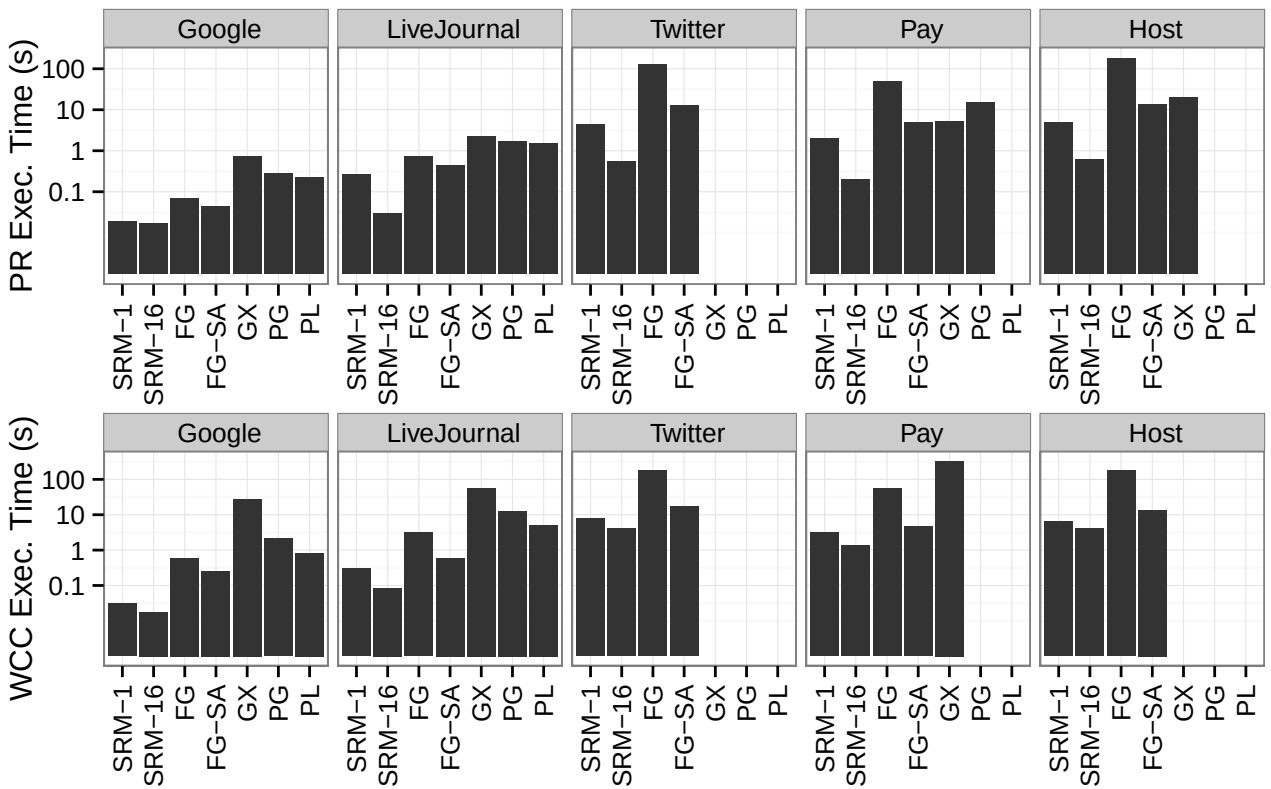
Ongoing work: PuLP in distributed memory

- Tentative distributed version of PuLP
- 20 iterations of PageRank on the web crawl with various partitioning strategies ($\frac{n}{p}$, $\frac{m}{p}$, random, PuLP)
- Using PuLP cuts best-case execution time in half



Comparison to popular frameworks

- Compared SRM-16 (Slota-Rajamanickam-Madduri) to GraphX (GX), PowerGraph (PG), and PowerLyra (PL) on 16 nodes of *Compton*; compared SRM-1 to FlashGraph (FG) and FlashGraph standalone (FG-SA) on a single node
- About 38× faster on average for PageRank (top), 201× faster for WCC (bottom) against distributed memory frameworks
- About 2.4× and 2.6× faster in shared-memory than FlashGraph

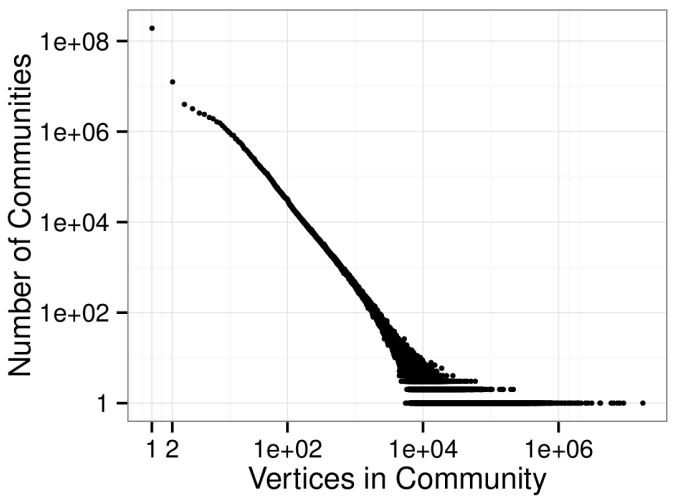


Community Detection in Web Crawl

- Used label propagation to study community structure of the web crawl
- Largest communities discovered after 30 iterations in table below
- Community frequencies from label propagation appear to follow a heavy-tailed power law

Largest Communities (numbers in millions)

Pages	In-Links	Out-Links	Rep. Page
112	2126	32	YouTube
18	548	277	Tumblr
9	516	84	Creative Commons
8	186	85	WordPress
7	57	83	Amazon
6	41	21	Flickr



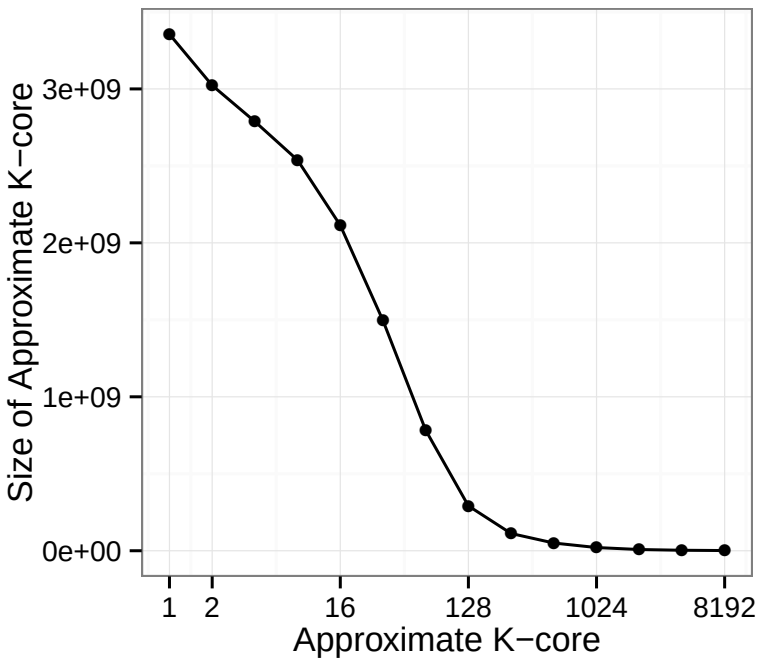
Centrality Measurements of Web Crawl

- Determined the top 10 web pages according to different centrality indices
- Similar to results found in prior work using smaller host-level graph [Meusel et al., 2014]
- Note that out-degree demonstrates very little importance as a centrality index

Out-degree	In-degree	PageRank	Harmonic Centrality
photoshare.ru/..	youtube.com	youtube.com	wordpress.org
dvderotik.com/..	wordpress.org	youtube.com/t/..	twitter.com
zoover.be/..	youtube.com/t/..	youtube.com/testtube	twitter.com/privacy
cran.r-project.org/..	youtube.com/..	youtube.com/t/..	twitter.com/about
cran.rakanu.com/..	youtube.com/t/..	youtube.com/t/..	twitter.com/tos
linkagogo.com/..	youtube.com/..	tumblr.com	twitter.com/account/..
cran.r-project.org/..	youtube.com/t/..	google.com/intl/en/..	twitter.com/account/..
fussballdaten.de/..	gmpg.org/xfn/11	wordpress.org	twitter.com/about/resources
fussballdaten.de/..	google.com	google.com/intl/..	twitter.com/login
fussballdaten.de/..	google.com/intl/..	google.com	twitter.com/about/contact

K-core Decomposition of Web Crawl

- Performed k-core decomposition using our approximation algorithm (gives upper-bound power-of-two)
- Plotted below is number of total pages in k-core versus k-core values
- K-cores appear quite large; ~300M pages in 128 k-core and ~20M pages in 1024 k-core



Future work

- Complete implementation of PuLP and DGL for communication and computation acceleration
- Continue to develop general purpose engine for many more graph analytics
- Use techniques to perform more in-depth studies of large and complex networks

Conclusions
Overall lessons learned

- Parallel algorithm design
  - Minimizing synchronization costs
  - Keeping memory accesses local
  - Even work distribution among threads and tasks
- Identifying algorithmic traits across graph algorithms
  - Many graph algorithms follow an iterative nested-loop structure
  - Many graph algorithms use common subroutines such as BFS, etc.
- Storing and organizing graphs efficiently in memory
  - Optimizing layout for specific graph types and applications
  - Balance cost tradeoffs for both communication and computation
  - Need for parameter tuning and experimental evaluation

Acknowledgments

- (2013-Present) – **Sandia and FASTMATH**
  - This research is supported by NSF grants CCF-1439057 and the DOE Office of Science through the FASTMath SciDAC Institute. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.
- (2014-2015) – **Blue Waters Fellowship**
  - This research is part of the Blue Waters sustained petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070, ACI-1238993, and ACI-1444747) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana Champaign and its National Center for Supercomputing Applications.
- (2012-2015) – **Kamesh Madduri's CAREER Award**
  - This research was also supported by NSF grant ACI-1253881.

Thank you! Questions?

Bibliography I

N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S.C. Sahinalp. Biomolecular network motif counting and discovery by color coding. *Bioinformatics*, 24(13):i241–i249, 2008.

H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 2014.

M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. Measuring user influence in Twitter: The million follower fallacy. In *Proc. Int'l. Conf. on Weblogs and Social Media (ICWSM)*, 2010.

Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proc. Int'l. Conf. on Data Mining (SDM)*, 2004.

A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel GPU methods for single-source shortest paths. In *Proc. IEEE Int'l. Parallel and Distributed Proc. Symp. (IPDPS)*, 2014.

Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. *ACM SIGPLAN Notices*, 46(8):267–276, 2011.

J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.

Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. Graph structure in the web - revisited: A trick of the heavy tail. In *Proc. WWW*, 2014.

Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. The graph structure in the web - analyzed on different aggregation levels. *J. Web Sci.*, 1(1):33–47, 2015.

U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

G. M. Slota, S. Rajamanickam, and K. Madduri. BFS and Coloring-based parallel algorithms for strongly connected components and related problems. In *Proc. Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, 2014.