



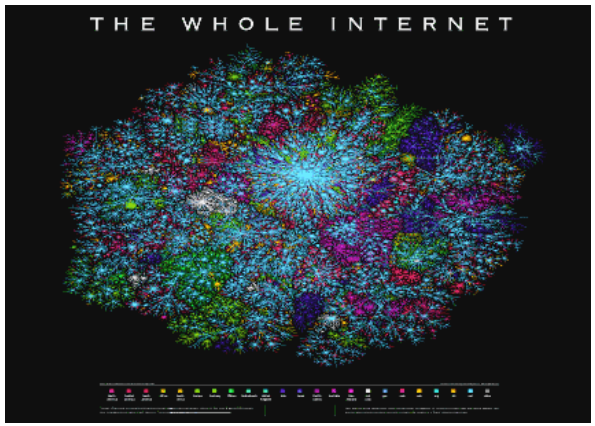

Optimizing Layout for Fast Graph Analysis

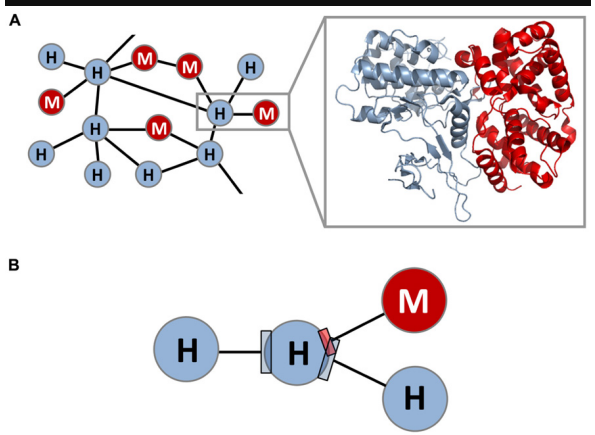
George M. Slota
Students: Christopher Brissette & Michael Mandulak
Rensselaer Polytechnic Institute

November 19th, 2021

Graphs are...

- **Everywhere**
  - Internet, information networks
  - Social, communication networks
  - Biology, chemistry
  - Scientific modeling, meshes, discrete interactions





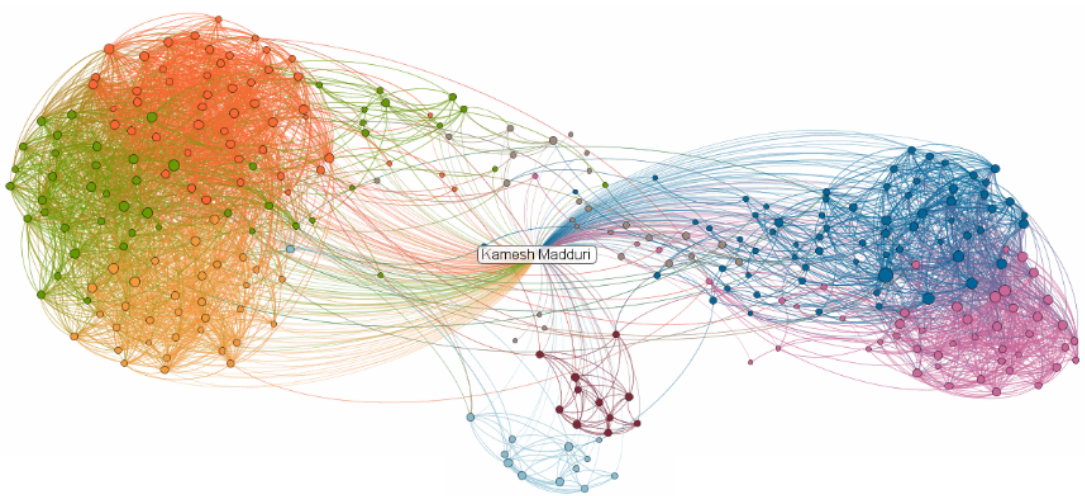


Figure sources: Franzosa et al. 2012, `http://www.unc.edu/ unclng/Internet_History.htm`

Graphs are...

- **Big**
  - Internet - 50B+ pages indexed by Google, trillions of hyperlinks (30T+ pages actually crawled)
  - Facebook - 2B+ users, 1T+ friendships
  - Human brain - 100B neurons, 200 – 1,000T synaptic connections





Figure sources: Facebook, Science Photo Library - PASIEKA via Getty Images

Graphs are...

- **Complex**
  - Graph analytics is listed as one of DARPA's 23 toughest mathematical challenges
  - Graphs are extremely variable - $O(2^{n^2})$ possible simple graph structures for n vertices
  - Real-world graph characteristics makes computational analytics tough

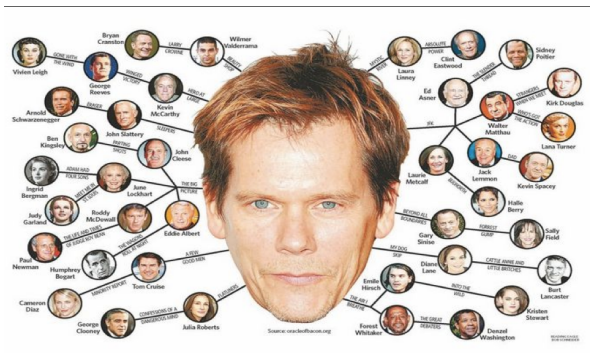


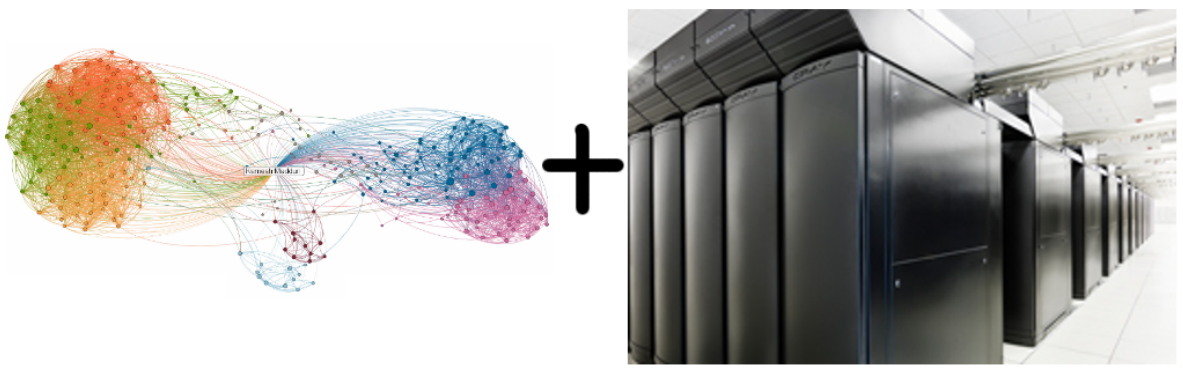
Figure source: `http://www.markrobinsonwrites.com/`

- Skewed degree distributions (power law, irregular)
- Small-world nature (i.e., *Six Degrees of Kevin Bacon*)
- Dynamically changing (optimizations that are effective today might not work tomorrow)
- Many interesting graph problems are NP-hard/complete, or otherwise intractable

HPC Systems are also...

HPC → High Performance Computing

- **Everywhere:** Multiple systems at most universities; recent widespread industry adoption (e.g., GPUs now common in data centers)
- **Big:** Thousands of nodes, millions of threads
- **Complex:** Hierarchical and heterogenous architectures with non-trivial network layouts



The Challenge...

So the challenge is:
How can we use HPC to enable the efficient solution to graph analytical or algorithmic problems?

We specifically observe three broad challenges:

- Communication costs for distributed processing are high
- General graph distributions can have poor cache utilization
- Limited accelerator memory exacerbates the above

Overview of this talk

We address these challenges in three parts:

Part 1: Graph partitioning for distributed GPU-based systems

- To address communication costs

Part 2: Vertex ordering for cache-efficient processing

- To address low cache utilization

Part 3: Hierarchical processing for GPUs and accelerators

- To optimize "out-of-core" processing for GPU+CPU

Note: All of our proposed approaches consider "graph layout" as the core problem.

Graph Layout

Graph Layout: The way a graph is stored is stored on a computational system. Typically, some variation of adjacency lists or a CSR (compressed sparse row) representation is used for basic storage, along with partitioning for distribution.

Specifically, we consider layout in terms of:

- Graph partitioning
- Vertex ordering
- Out-of-core and hierarchical processing

Each of these problems can often involve solving complex (NP-hard/complete) optimization problems. An optimal solution for one impacts the possible solution quality of others. E.g., an optimal partitioning might not lend itself to "good" per-part vertex orders.

Graph Partitioning

Reviewing the basic problem

Graph Partitioning (1D): Given a graph $G(V, E)$ and p processes or tasks, assign each task a p-way disjoint subset of vertices and their incident edges from G

- Balance constraints – (weighted) vertices per part
- Quality objectives – minimize (weighted) edge cut, communication volume, maximal per-part edge cut, etc.

Why?

- Extreme-scale datasets cannot reside in shared-memory of a single node
- Balance computation per rank, minimize inter-rank communication

Note: This is only a single formulation of the partitioning problem. Many alternatives exist. See Buluç et al. 2016.

Vertex Ordering

Also known as vertex "labeling"

Vertex Labeling: Assign to each vertex some unique orderable label, which is used in construction of graph data structure and processing.

- Typically, we use numeric values within $0 \ldots (|V| - 1)$
- Most graph datasets are given with some "natural" order (usually a result of origin crawling/parsing methodology)
- This natural order can usually be improved

Why would we do this?

- Cache efficiency for computations, compressibility, and algorithmic overheads are directly impacted
- Ordering can be lightweight and greatly improve computation times

Architecture-centric Processing

Hierarchical and out-of-core processing

Architecture-centric Processing: Designing processing patterns for graph data that best utilize the underlying hardware architecture. We explicitly consider two variations.

Hierarchical Processing:

- Consider the hierarchy: storage, network, CPU, accelerators
- Any proposed solutions to our layout problems *need* to take this underlying hierarchy into account

Out-of-Core Processing:

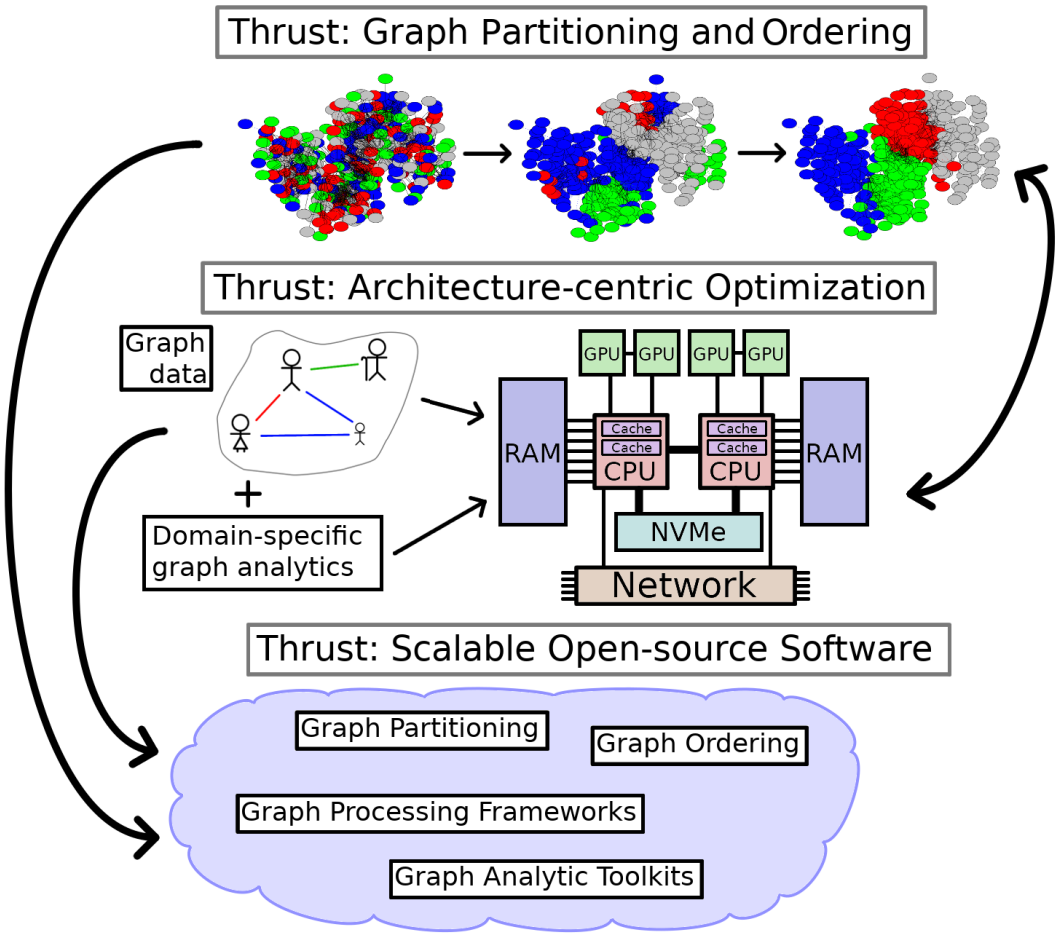
- Accelerators have limited memory relative to the full node
- We can store large portions of the graph on-node, but only process subparts on acclerators at a time
- The tradeoff: wider distributed parallelism (high communication costs) vs. (psuedo-)out-of-core processing for accelerators (higher compute times)

The BIG problem

Everything is inter-related!

We consider the broad and general problem of:

Mapping graph data→layout→architecture-centric processing
→scalable software to address these problems



Part 1: Graph Partitioning

Ongoing work with CAREER and FASTMath projects.
Includes work from Ph.D. student Chris Brissette.

The common approach

But definitely not the *only* approach

Most graph partitioners use a "multilevel" approach.

- First, *coarsen* an input graph to a smaller representation
- Next, *initialize* some partition on this coarsened graph
- Then, *uncoarsen* and *refine* this partition to the full graph

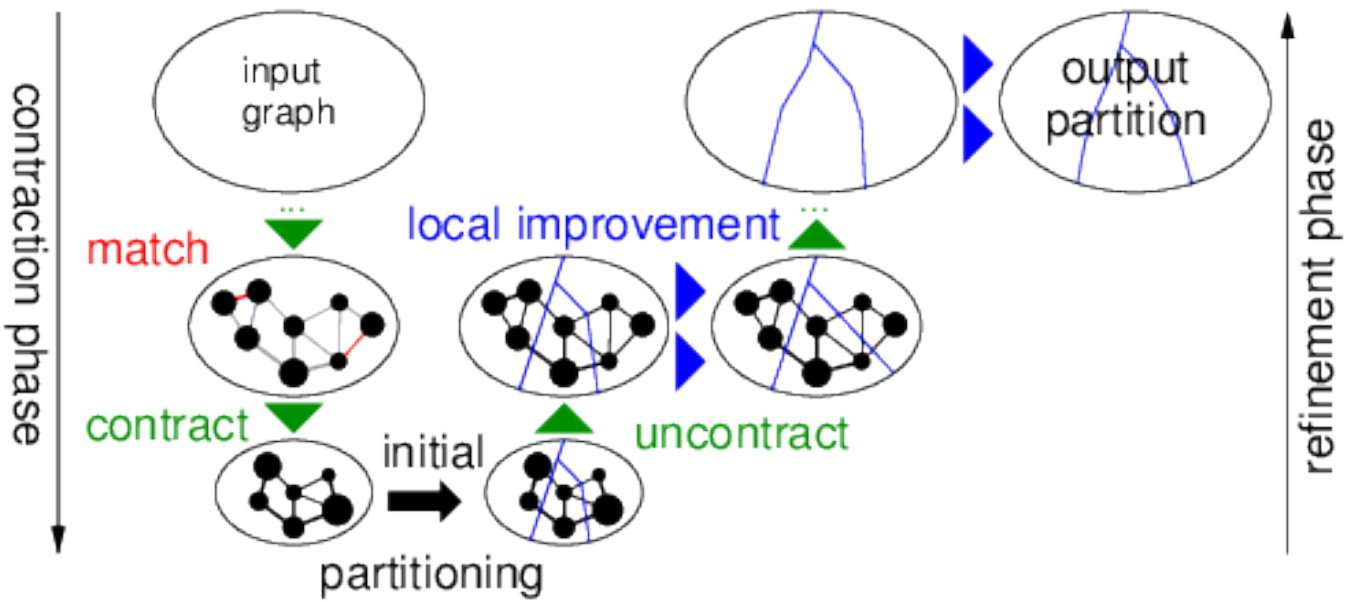


Figure source: Meyerhenke et al., 2014

Prior Work

Many existing software tools:

- METIS, Scotch, KaHIP, Zoltan, ParMA, PaToH, PuLP/XtraPuLP, etc.

Many coarsening approaches:

- Heavy-edge matching, Heavy-edge coarsening, MIS2, GOSH, etc.

The big considerations:

- Processing time and parallelization strategy
- Memory overheads
- Output quality - benefit to actual application

For processing extreme-scale graph datasets, **graph coarsening and associtation memory overheads of storing intermediate graph representations is a key consideration.**

Our Overall Partitioning Strategy

Overview of our partitioning strategy:

- Near-constant memory parallel coarsening
  - *Why:* reduce overheads for extreme-scale graphs
  - Ongoing: retain spectral properties
- Low-level spectral partitioning of coarse graph
  - *Why:* easy and fast to distribute on GPUs
- Efficient refinement and uncoarsening procedure
  - *Why:* Usually one of the costliest procedures

Near-constant memory coarsening

The 'near' for when it's distributed

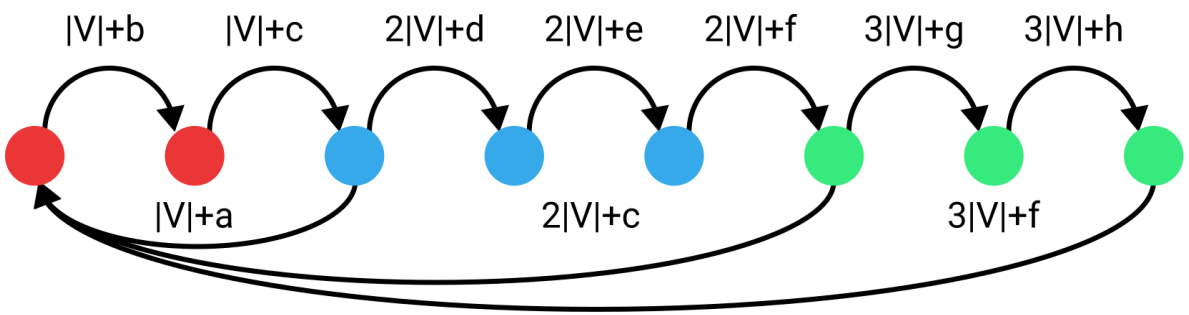
Want to perform multilevel coarsening on a graph $G = (V, E)$ with constant memory overhead. We can do this using an array of size $2(|V| + |E|)$.

a. Start with a CSR adjacency list with $2|E|$ entries.

b. For each node v add an entry $|V|r + u$. r is the level of coarsening that v absorbed a node and u is the node that was absorbed by v.

c. For each node v add an entry $|V|k + u$. k is the level of coarsening that v was absorbed and u is the node that absorbed v.

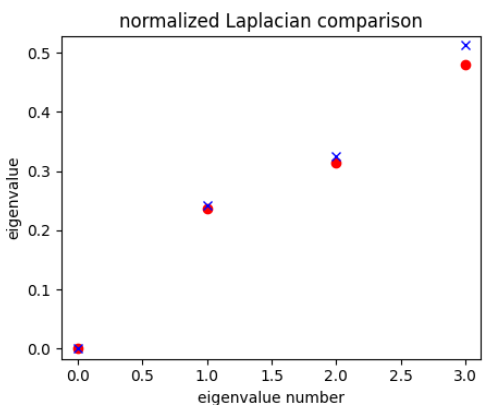
d. The first value acts as a pointer to continue the node adjacency of subsuming nodes. The other determines what node edges point to in the k^{th} level of coarsening.

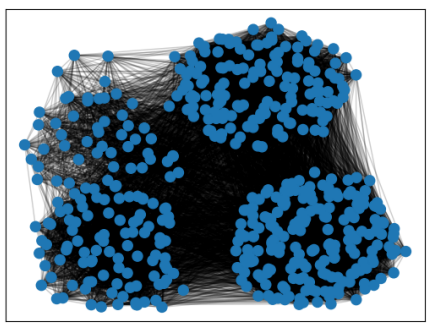


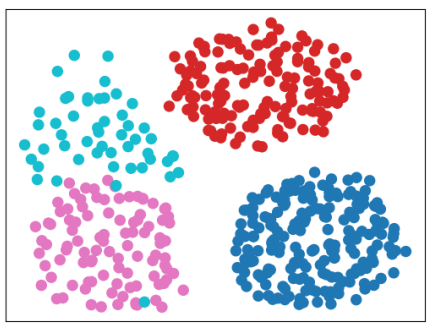
Ongoing: Random walk based spectral coarsening

If $\|p_k(u,v) - p_k(u,x)\| \leq \epsilon$ for each pair of merged nodes v, x and each random walk length k, then the eigenvalues of $\mathcal{L}(G_c)$ and the leading eigenvalues of $\mathcal{L}(G)$ are ϵ approximations of eachother.

1. Perform random walks of length k from a sample of nodes $S \subseteq V$ and track the nodes visited as a probability vector.
2. Partition our sample into C clusters, maximizing the random walk similarity between nodes in each cluster.
3. Greedily add each node to the cluster it is the most similar to with respect to random walk vectors.







Common Partitioning Myths

When performaning spectral partitioning

Myth: Spectral partitioning is *slow*
Fact: Spectral partitioning can actually be faster than optimized single-level methods:

- See Seher et al., 2020
- Spectral cuts with MultiJagged and Trilinos solvers

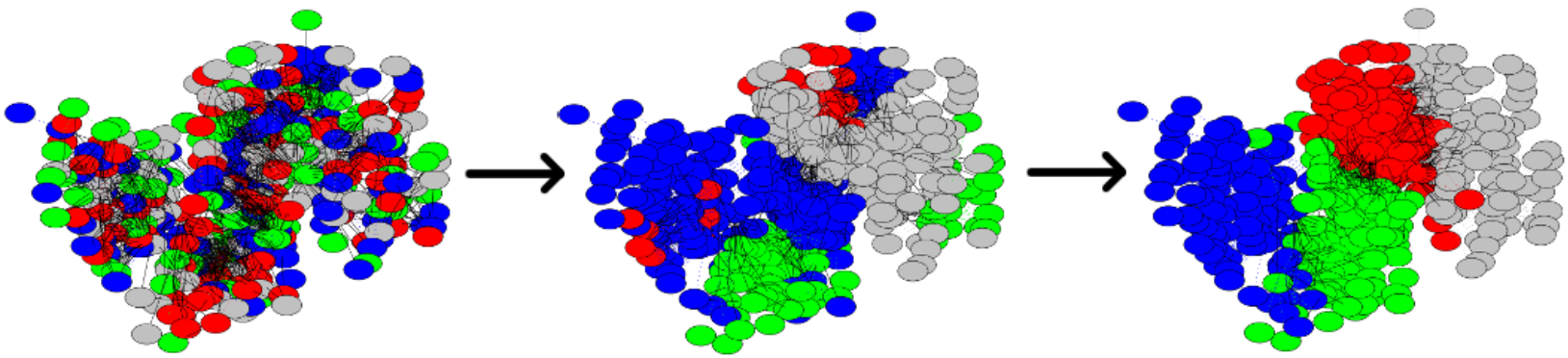
Myth: Spectral partitioning is *low quality*
Fact: When combined with refinement in a multi-level framework, quality is as good as, or better than, other state-of-the-art approaches

- See Gilbert et al., 2021 and this current work

XtraPuLP-based refinement

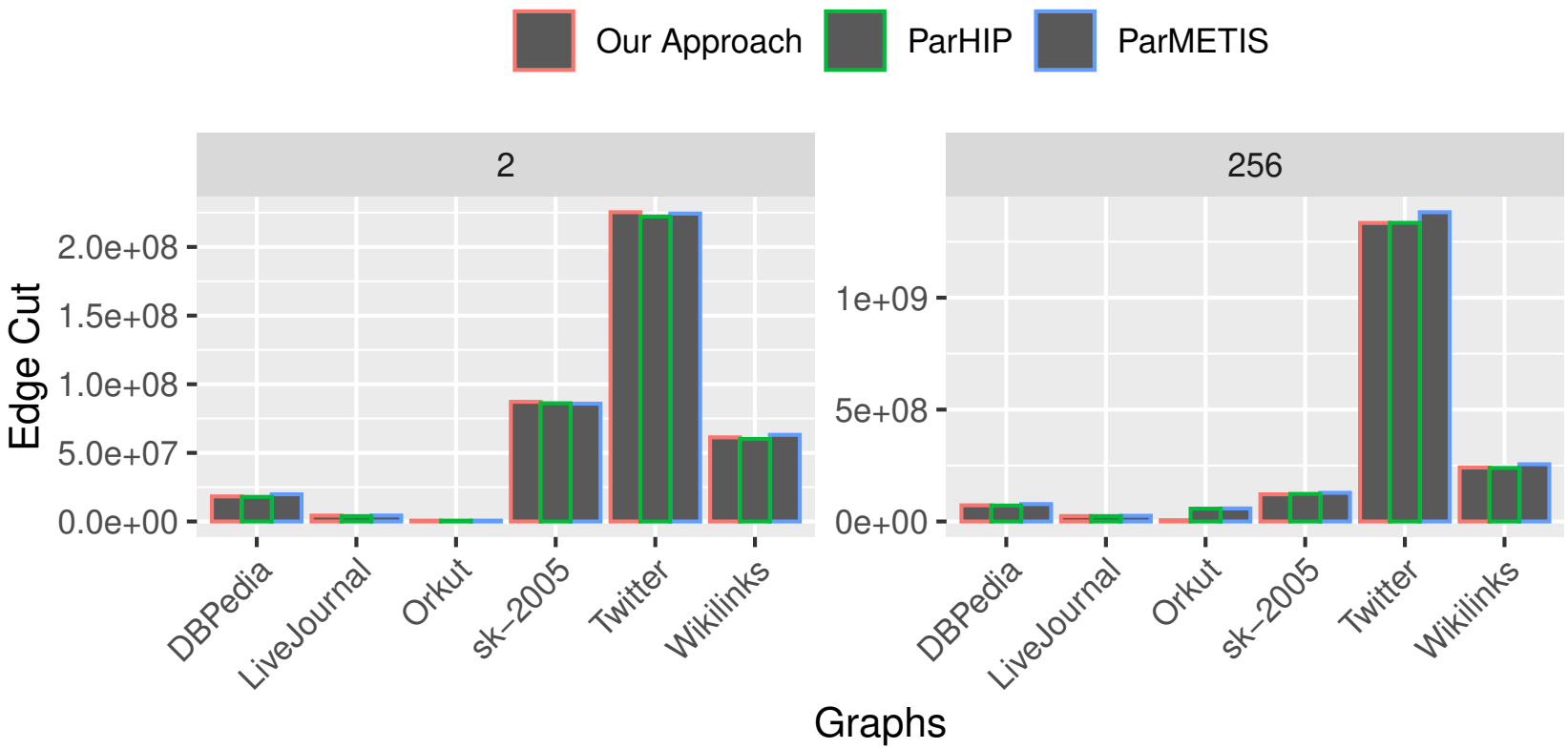
We utlize our prior partitioners, PuLP/XtraPuLP, to perform partition refinement.

- Utilizes a weighted variation of Fiduccia-Mattheyses (FM) refinement to balance and improve cut
- Highly efficient in both shared and distributed memory
- Ongoing: G-PuLP implementation for GPUs



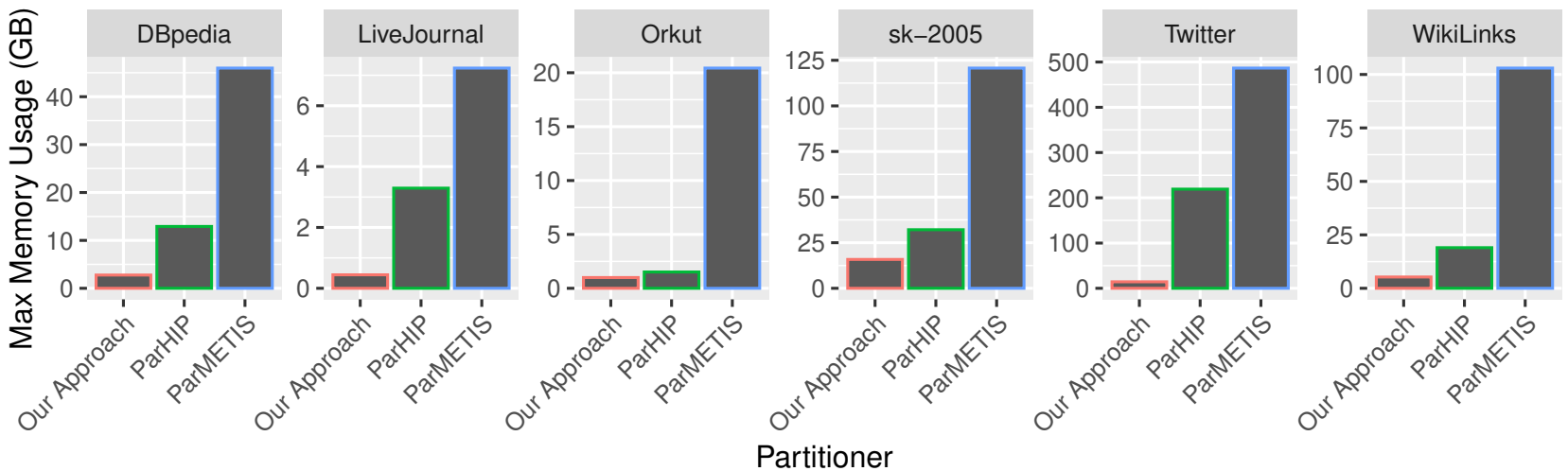
Partitioning Quality

Our approach gives partitioning quality in-line with other methods. Here we run with 256-way parallelization and compute 2 and 256 parts.



Maximum Memory Utilization

We compare our approach to ParMETIS and ParHIP in terms of maximum memory utilization. 256-way parallelization to compute 256 parts is used for the comparison.



Future Work

This work is currently ongoing, we have several obvious continuing avenues:

- Fully optimize GPU implementation
- Extend the general approach to 2D hierarchical partitioning
- Improve scalability of the distributed coarsening phase
- Improve quality of refinement and balance for GPUs

Part 2: Vertex Ordering

Ongoing work with CAREER project. Includes work from Ph.D. student Michael Mandulak.

Ordering: Prior Work

Most prior work falls into three categories:

Optimizing for fill reduction:

- RCM, nested dissection, bandwidth reduction, minimum degree
- Generally reduce "fill" in iterative sparse matrix computations

Optimizing for compression:

- WebGraph Framework (LLP), Shingle ordering, Slashburn
- Generally reduce *distances* between vertex labels

Optimizing for cache efficiency:

- Rabbit Order, BFS/DFS, many of the above algorithms
- Generally use clustering-based approaches, often without an explicit optimization objective

We consider an explicit combination of the last two categories, *we explicitly optimize for some objective to improve cache efficiency of graph computations*.

Graph Arrangement Measures

Consider graph $G = (V, E)$. $L(v)$ is to be the *unique* label of vertex $v \in V$ on $0 \ldots (|V| - 1)$. We consider minimizing the following metrics:

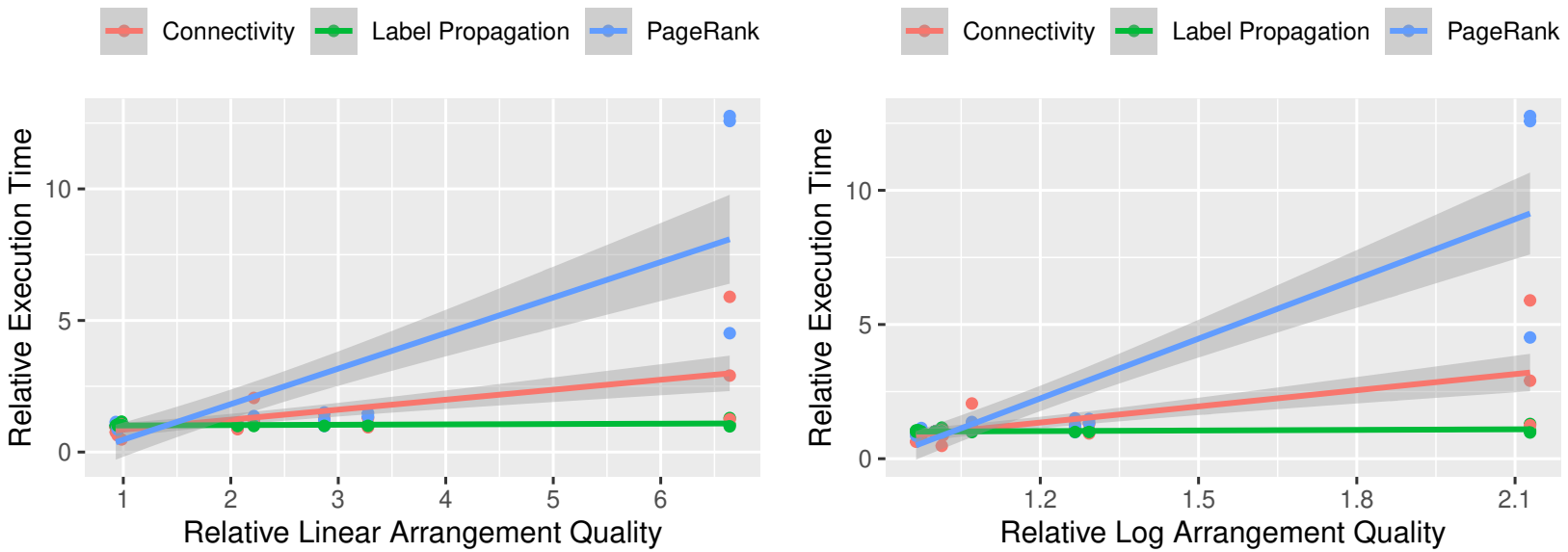
Linear Arrangement:
$\sum_{\forall v \in V} |L(v) - L(u)| : u \in N(v)$

Log Arrangement:
$\sum_{\forall v \in V} \log |L(v) - L(u)| : u \in N(v)$

Note: These aren't exactly representative of cache utilization, but close enough (for now). They are also straightforward to optimize within our optimization framework.

Correlating measure quality to execution time

We naively re-order our vertices using common ordering approaches. We observe a notable correlation between measurement quality and execution time.



Test graphs: Friendster (SNAP), it-2004 (WebGraph), Erdös-Rényi
Test Algorithms: Multistep Connectivity, Label Propagation, PageRank
Orderings: Natural, Random, BFS, RCM
Parallelization: 1, 128, 256 threads

Explicit Refinement

We consider an approach to improving ordering quality similar to those used in refinement of graph partitioning. Our approach is a variation of Kernighan-Lin (KL) Refinement.

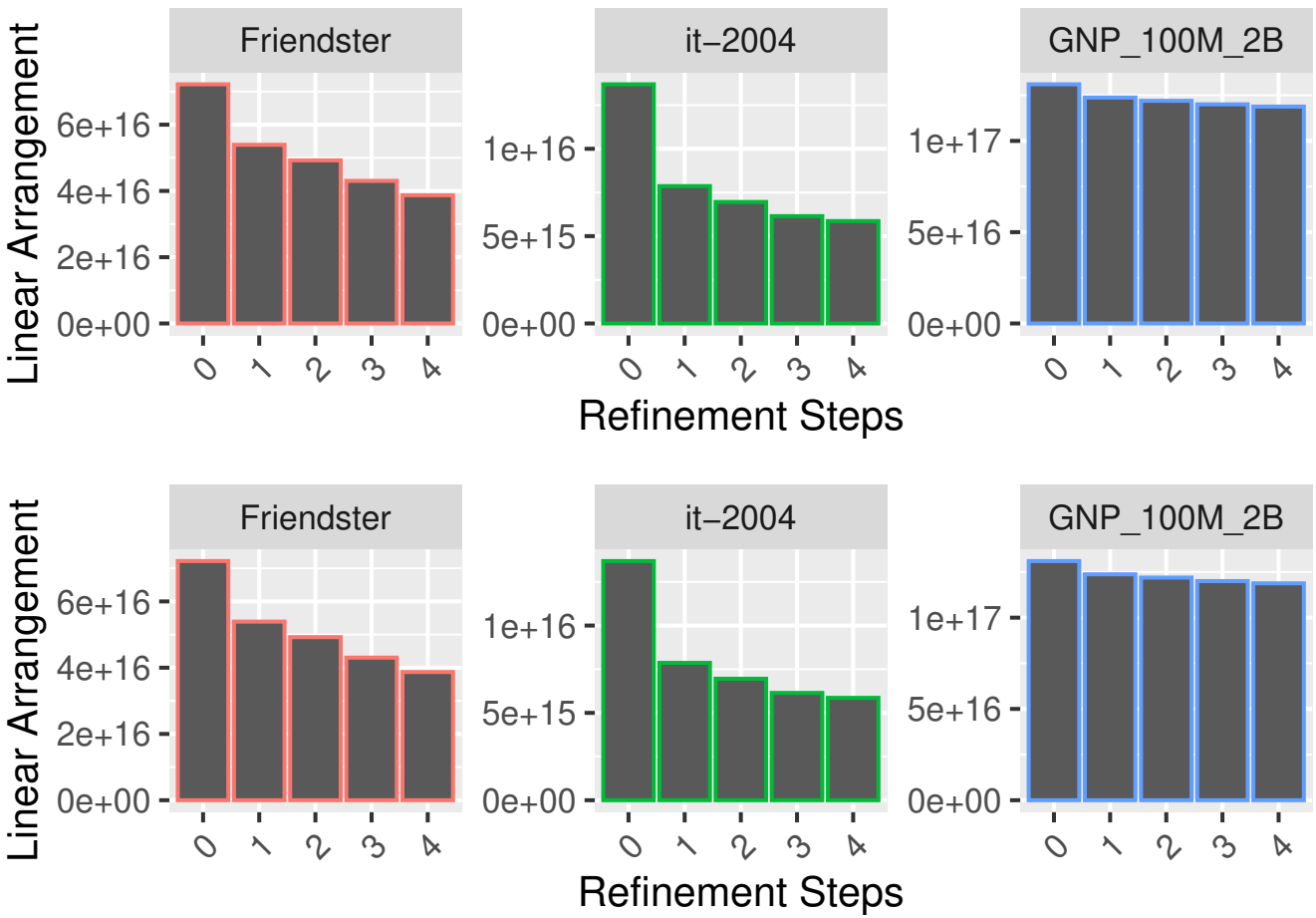
- Specifically, we consider potential *swap pairs* of vertex labels
- We swap these labels when they improve our overall metric quality

How do we select which pairs to swap?

- Evaluating *all* possibilities can be slow $O(|E|d_{max}^2)$
- We can limit potential swap partners per vertex
- We can also perform random walks, swapping as we go

Refinement Works!

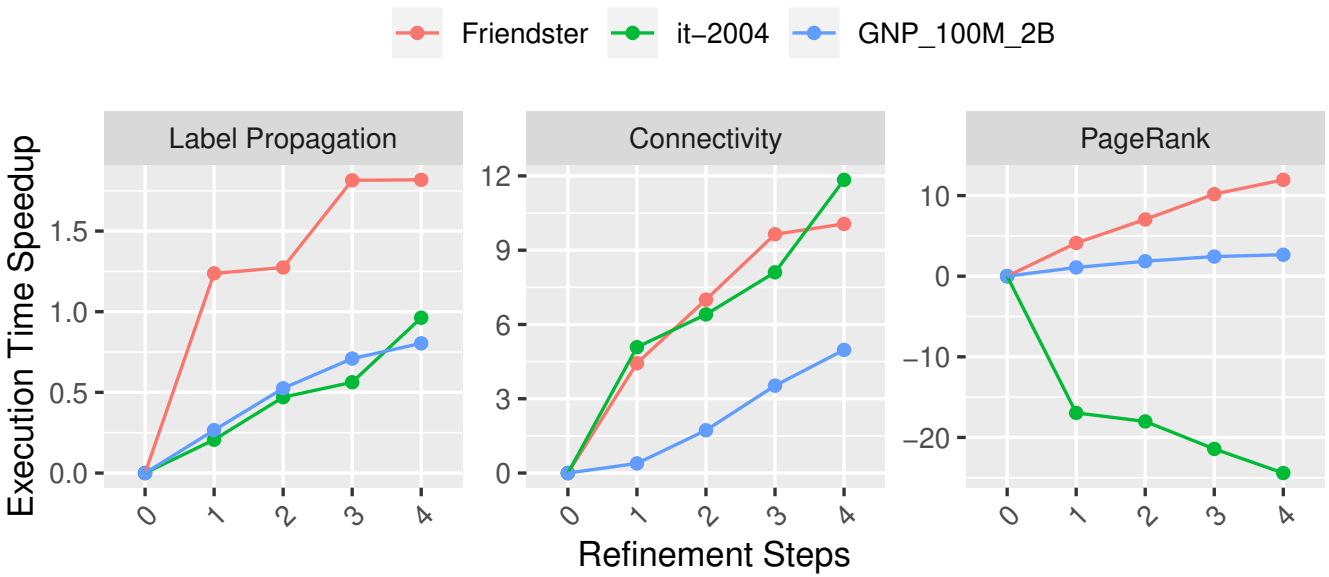
We can improve the ordering quality of an arbitrary initial ordering. Here, we refine RCM w.r.t. linear and log arrangements using label swaps.



Execution time vs. Refinement

Refinement also works to improve execution time!

We observe a correlation between refinement and execution time improvements. We show execution time speedup vs. refinement for our test algorithms and the log arrangement measure.



However, as *it-2004* on PageRank demonstrates, this relation can be complex, as our measures are not *exactly* representative of cache usage.

Future Work

We're considering several avenues for future work:

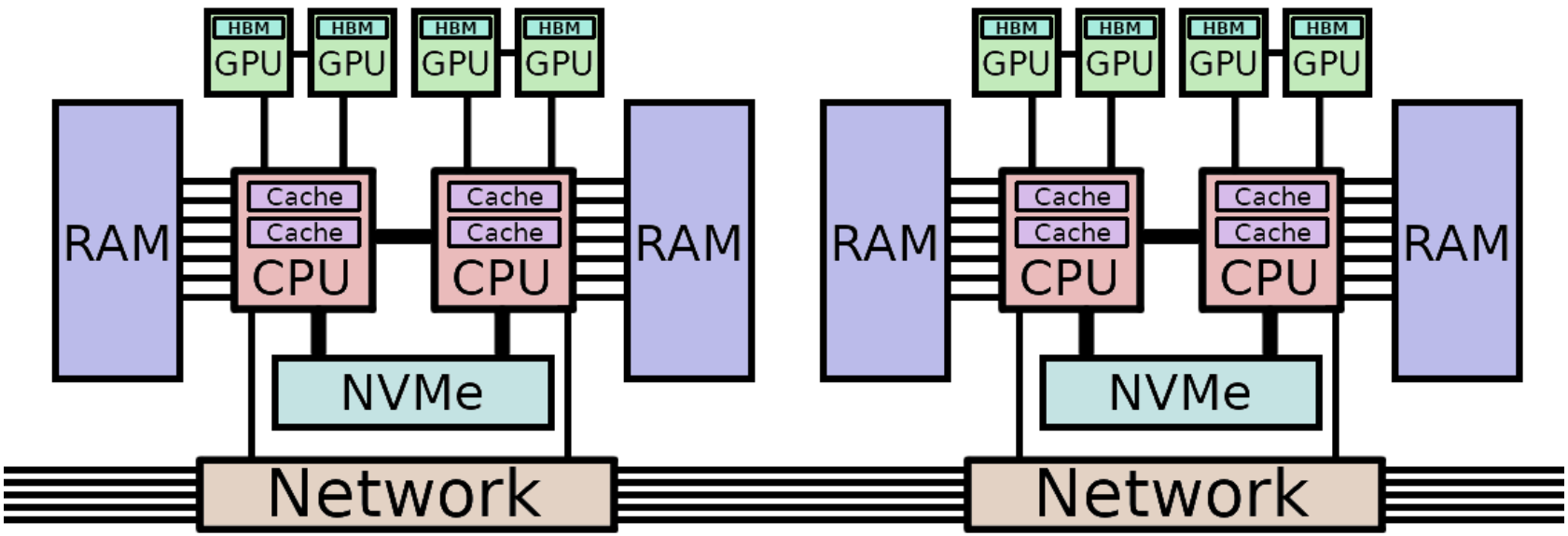
- Explore the interplay with partitioning methodology
- Explicitly optimize for cache utilization
- Optimize timing for our refinement procedure
- Implement on GPUs for further acceleration

Part 3: Graph Processing

Prior work with Sandia LDRD project and ongoing work with CAREER project. Published previously in IPDPS 2018.

The compute hierarchy on a modern HPC system

Most modern compute nodes have CPUs, accelerators with limited HBM (high-bandwidth memory), large DDR, local non-volatile storage, and a network connection to other nodes and mass storage.



Compute hierarchy: Continued

Ongoing trend: expansion of memory hierarchy for increased computational throughput via CPUs and accelerators

- E.g., high-bandwidth memory (HBM) on accelerators, SSD or NVRAM storage on nodes, multiple levels of CPU cache and GPU cache/shared memory

Can we explicitly design graph computations to effectively utilize these layers?

- We explore a work chunking approach that iteratively brings in pieces of a large graph to perform local updates in HBM – we specifically look at the *label propagation* algorithm. We find:
  - **Chunking has minimal impact on solution quality**
  - **Chunking can also decrease time to solution**

Primary assumption: the graphs being processed are too large to fit entirely within high bandwidth memory

Multilevel Memory Label Propagation

via work chunking

1: $L \leftarrow$ LPChunking$(G(V, E), \mathbf{C}_{num}, \mathbf{C}_{iter})$
2: **for all** $v \in V$: $L(v) \leftarrow id(v)$ ▷ Initialize labels as vertex ids
3: **while** *at least one L(v) updates* **do**
4: **for** $c = 1 \cdots C_{num}$ **do**
5: $V_c \leftarrow$ **Chunk**(c, V), $E_c \leftarrow \langle v, u \rangle \in E : v$ **or** $u \in V_c$
6: **for** $iter = 1 \ldots C_{iter}$ **while** *one* $L(v) : v \in V_c$ *updates* **do**
7: **for all** $v \in \mathbf{V_c}$ **do in parallel** ▷ Random order
8: *Counts* $\leftarrow \emptyset$ ▷ Hash table
9: **for all** $\langle v, u \rangle \in E_c$ **do**
10: *Counts*$(L(u)) \leftarrow$ *Counts*$(L(u)) + 1$
11: *NewLabel* $\leftarrow$ GetKeyOfMaxVal(*Counts*$(\ldots)$)
12: **if** *NewLabel* $\neq L(v)$ **then**
13: $L(v) \leftarrow$ *NewLabel*

Chunking Considerations

Primary chunking variables

- Number of total chunks (C_{num})
- Work iterations performed on each chunk (C_{iter})

How to determine data per chunk?

- Block methods (vertex block, edge block)
- Randomization or hashing
- Explicit partitioning

How to transfer chunked data?

- All CPU threads handle transfers on multi-GPU systems
- All CPU threads transfer, then all threads help with work
- Vary transfer+work threads dynamically

Experimental Setup

Test System and test graphs

Network	n	m	$davg$	$dmax$	$\tilde{D}$
LiveJournal	4.8 M	69 M	18	20 K	18
Friendster	66 M	1.8 B	27	5.2 K	34
Twitter	52 M	2.0 B	37	3.7 M	19
Host	89 M	2.0 B	22	3.4 M	23
uk-2007	105 M	3.3 B	31	975 K	82
wBTER_50	50 M	1.2 B	24	110 K	12
wBTER_100	100 M	2.4 B	24	135 K	12

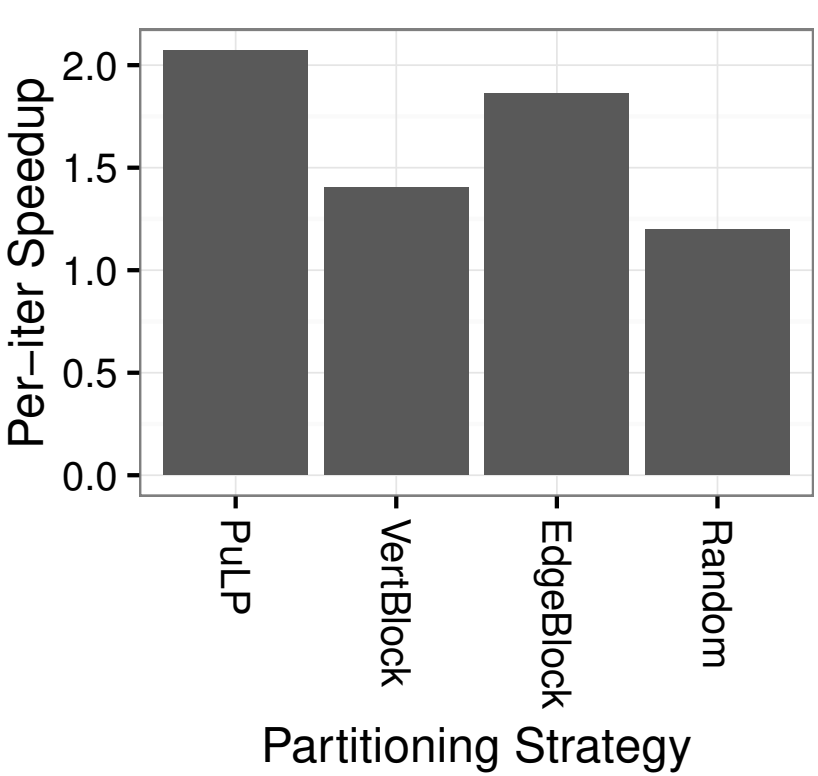
Test System 1: *Zepy* at RPI – 4xA100 GPU with 40GB GDDR and 512GB DDR

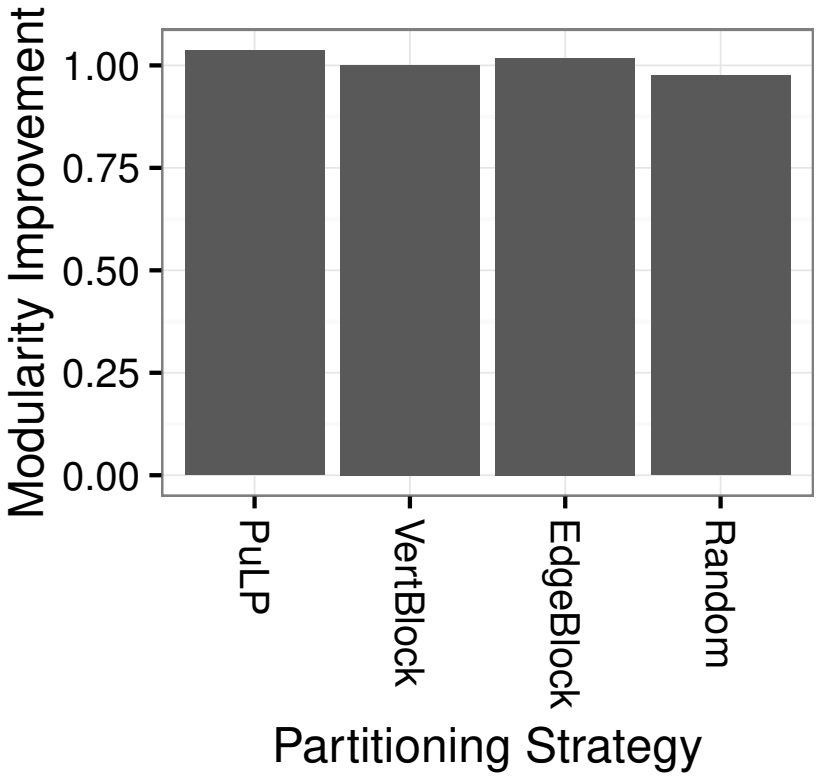
Test System 2: *Bowman* at Sandia Labs – Intel KNL with 68 cores, 96 GB DDR, and 16 GB HBM

Effect of Partitioning Methodology

5 iterations per chunk, minimum number of chunks possible (~5), 40 iterations

- Effects of partitioning method on per-iteration speedup vs. baseline timing (left) and modularity (right)
- Explicit partitioning demonstrates largest improvements, but at the obvious cost of computing the partition

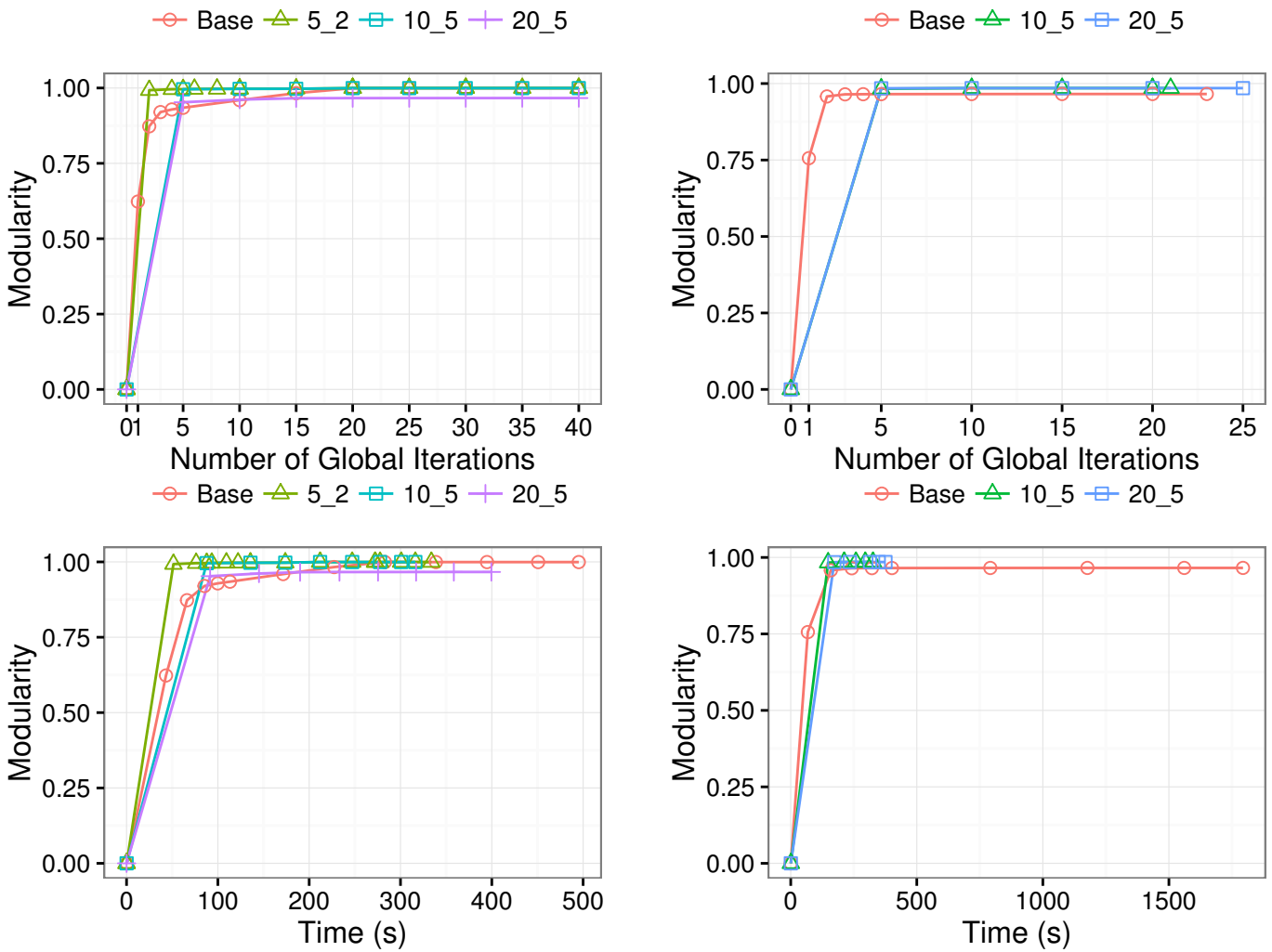




Time and modularity vs. iterations

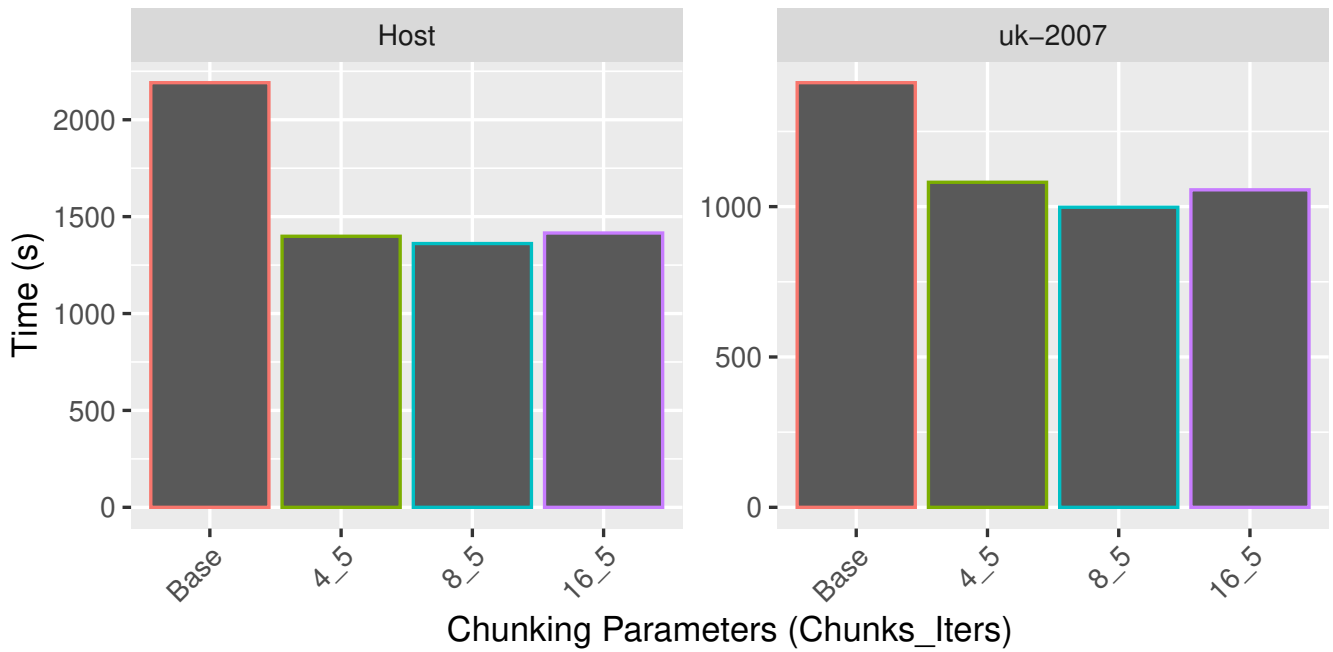
Per-iteration time and total time doesn't tell the whole story

Friendster (left) and Twitter (right) for modularity vs. iterations (top) and time per iteration (bottom). Baseline and $C_{num}_C_{iter}$.



Utilizing the hierarchy

We extending our chunking approach to multi-GPU nodes. We partition the input into HBM-sized chunks. Our CPU handles partitioning, data transfer, and updates to algorithmic data. The GPUs perform algorithmic work. Edge block is used.



Conclusions

and future work

- Chunking minimally affects solution quality of label propagation, but can increase the number of iterations required for a given "quality"
- Explicit handling of HBM generally improves per-iteration timing and can improve time-to-solution in select instances
- Future work:
  - Further explore interplay with partitioning and ordering
  - Further explore generalizations to other vertex programs
  - Multi-tiered chunking – hold key vertices in HBM and update every iteration

Acknowledgements

This work was supported by the following awards:

- CAREER: Scalable Graph Processing to a Quadrillion Edges and Beyond, National Science Foundation, Award #2047821.
- Frameworks, Algorithms and Scalable Technologies for Mathematics (FASTMath) SciDAC-5 Institute, Department of Energy Office of Science.
- Multi-Level Memory Algorithmics for Large, Sparse Problems, Sandia National Labs[1] Laboratory Directed Research and Development (LDRD) Project.
- MRI: Acquisition of a Next Generation, Data-Centric Supercomputer, National Science Foundation, Award #1828083.

[1] Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

Conculsions

and Thanks!

Main takeaway: layout is a key consideration for graph analytics

- Multilevel memory partitioning *does not* need to have high overheads
- Refinement of vertex ordering can lead to over a 10% improvement in execution time over existing ordering approaches
- Work chunking for multilevel memory can have *significant* benefits to both time to solution as well as final quality for community detection.

Contact: `gmslota@gmail.com`, `www.gmslota.com`