Rensselaer

Sandia National Laboratories

Scalable Partitioning on GPU

George M. Slota

Rensselaer Polytechnic Institute

SIAM Annual Meeting 2020

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.

Overview
Scalable Partitioning on GPU

Big Idea: Implement (Xtra)PuLP partitioners for GPU

- Using Kokkos for performance portability
- Currently: 1x GPU (G-PuLP); Future: Many GPUs (G-XtraPuLP)

Challenges: Typical for many CPU $\rightarrow$ GPU ports

- Fine-grained parallelization of thread-based processing
- Mitigating effects of PuLP's asynchronous processing

Outcomes:

- Can be faster – effective fine-grained parallelization
- But worse quality – difficult to mitigate asynchronicity
- However: Exploring "tuning" methods to allow tradeoffs of quality vs. speed

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

Why?

- Processing patterns of many distributed scientific computations can be represented as a graph
- Balance computation per rank, minimize inter-rank communication

PuLP and Other Partitioning Software

Many existing software tools: METIS, Scotch, KaHIP, Zoltan, ParMA, PaToH, etc.

- Each tool has various tradeoffs
- E.g., processing speed, possible constraints/objectives, output quality, target graph structures

(Xtra)PuLP – Partitioning Using Label Propagation

- Uses diffusive label propagation-based approach
- Highly optimized for scalability (1 trillion+ edge graphs)
- Targets partitioning problem:
  - Multiple constraints – some number of vertex weights
  - Multiple objectives – minimize total communication and balance communication load per-rank
  - On highly irregular very large-scale inputs

GPU Processing

Why GPU?

- Targeting current and next-generation heterogeneous platforms
- Ideally, run partitioning/pre-processing on same hardware as target applications

AiMOS at RPI's Center for Computational Innovations

Methodology
Modifying PuLP for GPU

We have two major considers for implementing PuLP on GPU.

1. **Fine-grained parallelism**
   - CPU = ~dozens of cores, GPU = ~thousands of cores
   - In particular, this presents a number of thread-based *work imbalance issues* for graph problems on irregular datasets
2. **Asynchronous processing**
   - PuLP relies on asynchronous updates for speed (and quality)
   - On GPU, this results in an *orders-of-magnitude increase* in computations on "stale" data

Fine-grained Parallelism

Consider: regular graph vs. highly irregular graph

- Regular graph has fixed degree distribution, irregular graphs can be extremely skewed
- *Vertex-centric* parallelism quickly "breaks down"

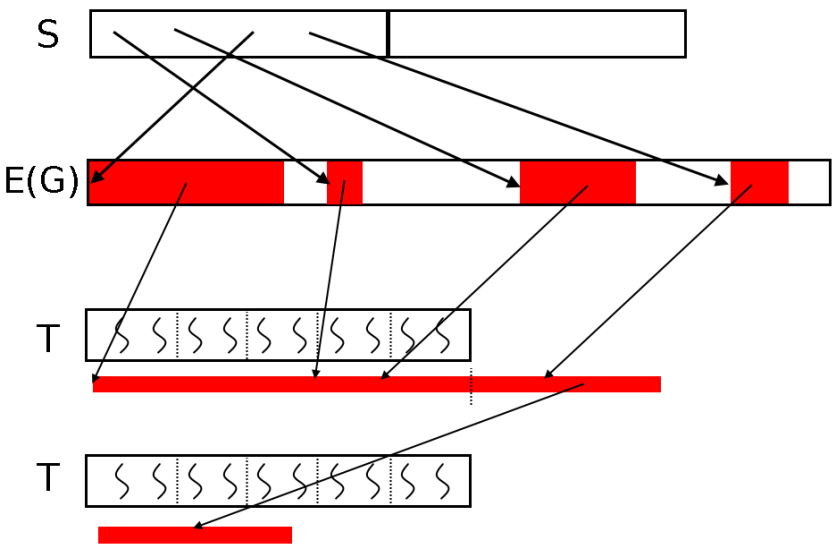
There are many proposed techniques in the literature to overcome this challenge:

- Hierarchical parallelism (Hong et al., PPoPP 2011)
- Graph structure modification (e.g., "SlimCell" by Besta et al., IPDPS 2017)
- **Loop Collapse** (Slota et al., IPDPS 2015)

We use a loop collapse method because it is fully parallelizable, requires no modification to the graph adjacency structure, and can be used with arbitrary workloads such as queues.

Loop Collapse

Manhattan Collapse: Optimization technique for nested loops (possibly) first appearing in the literature for Cray XMT systems[1].



- Many graph-based algorithms follow a nested loop structure
  – For some set of vertices S (outer loop), process all of their adjacencies from the global adjacency list $E(G)$ (inner loop)
- The Manhattan Collapse assigns each unit of inner loop work to a given thread t within a thread block T
  – **Regardless of the order of items in the outer loop!**

[1] Ringenburg and Choi, 2009

Parallelism of PuLP for CPU

Representative subroutine of processing pattern

The original processing pattern of the majority of PuLP's subroutines when parallelized on CPU is below.

1: **procedure** PuLP-Balance($G(V, E), parts$)
2: $Q \leftarrow V(G)$ ▷ initialize queue with all vertices
3: $Q_{next} \leftarrow \emptyset$
4: **for** some number of iterations **do**
5: **for all** $u \in Q$ **do in parallel**
6: $C(1 \ldots p) \leftarrow 0$ ▷ neighboring part counts
7: **for all** $e = \{u, v\} \in E(G)$ **do**
8: $C(parts(v)) \leftarrow C(parts(v)) + 1$ ▷ enumerate all part counts
9: $gains \leftarrow$ ComputeGains(C) ▷ per-part gain of moving v to each p
10: $p' \leftarrow$ Max($gains(1 \ldots p)$) ▷ best part assignment for v
11: **if** $p' \neq parts(v)$ **then**
12: $parts(v) \leftarrow p'$ ▷ update v's part assignment
13: $Q_{next} \leftarrow \{v, \forall e \in N(v)\}$ ▷ add v and its neighbors to queue
14: Swap(Q, Q_{next})

Parallelism of G-PuLP for GPU

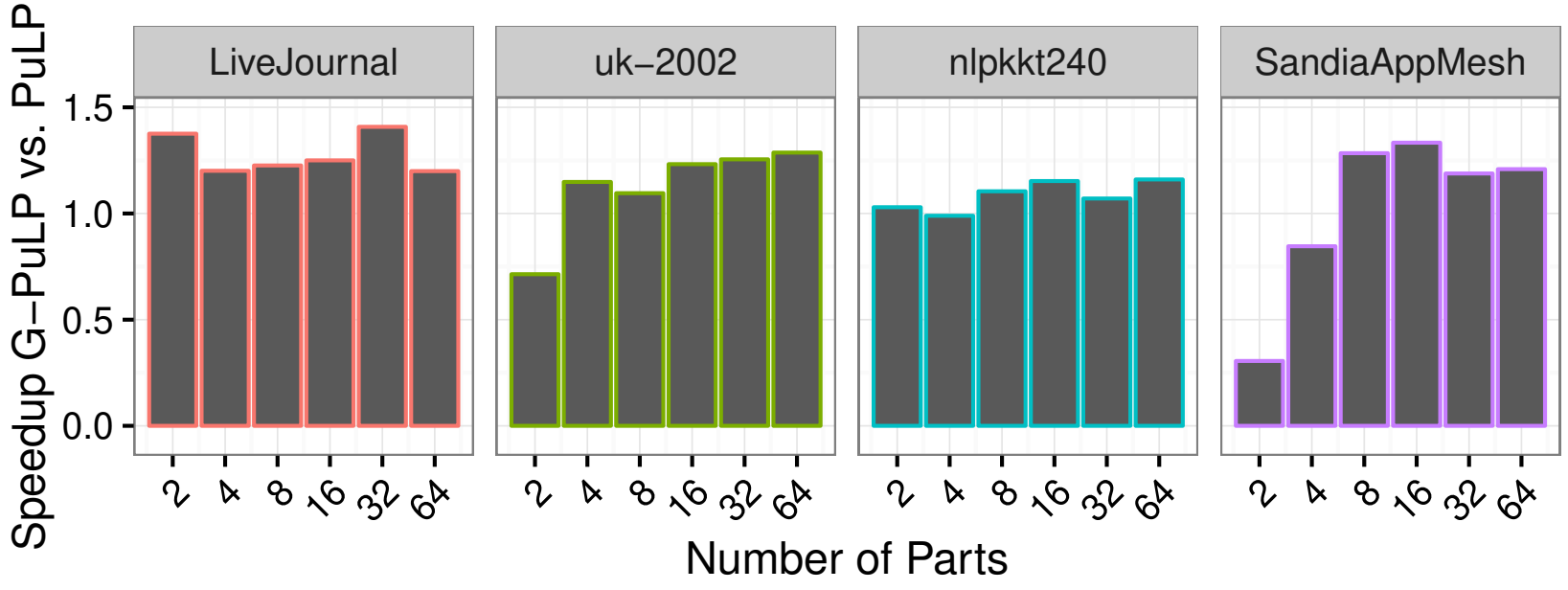
Modifications to run on GPU

1: **procedure** G-PuLP-Balance($G(V, E), parts$)
2: $Q \leftarrow V(G)$ ▷ initialize queue with all vertices
3: $Q_{next} \leftarrow \emptyset$
4: **for** some number of iterations **do**
5: **Launch Kernel:** (thread block T with threads t)
6: $S' \leftarrow$ vertices assigned to T ▷ Note: $|S'| = |T|$
7: $C_t(1 \ldots p) \leftarrow 0$ **in parallel** $\forall t \in T$ ▷ per-thread counts
8: $W \leftarrow$ PrefixSums$(N(v) : \forall v \in S')$ ▷ total work for team
9: **for all** $w \in W$ **do in parallel** $\forall t \in T$
10: $v \leftarrow$ **GetVert**(w, S') ▷ Using Manhattan collapse
11: $u \leftarrow$ **GetEdge**(w, S', v)
12: $C_t(parts(u)) \leftarrow C_t(parts(u)) + 1$
13: $C \leftarrow$ Reduce(C_t) **in parallel** $\forall t \in T$
14: $gains \leftarrow$ ComputeGains(C) **in parallel** $\forall t \in T$
15: $parts(S') \leftarrow$ UpdateParts$(gains)$ **in parallel** $\forall t \in T$
16: $Q_{next} \leftarrow$ AddToQueue(S') **in parallel** $\forall t \in T$
17: **End Kernel**
18: Swap(Q, Q_{next})

And we see some pretty good performance

Yay!

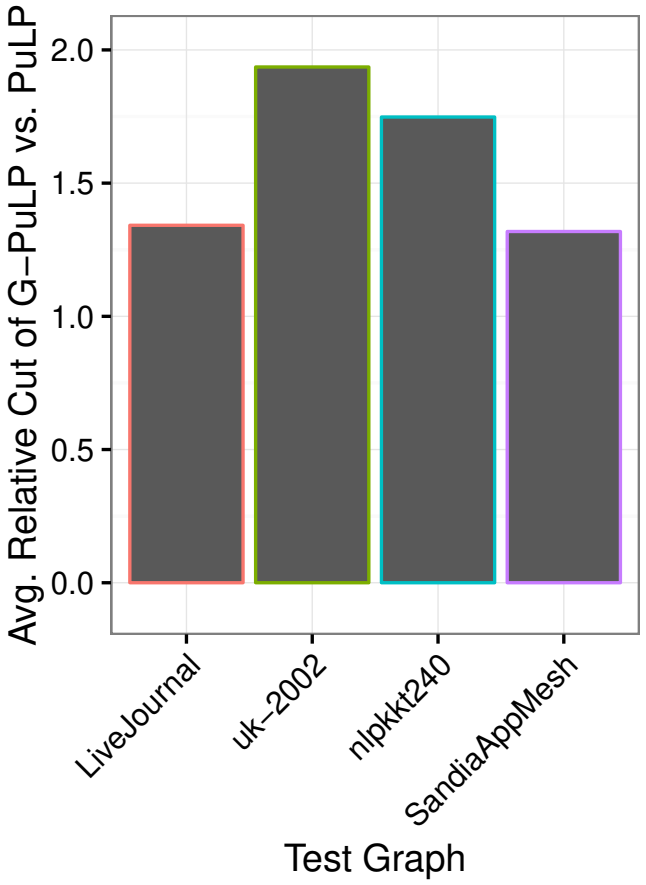
We implement the preceding approach using Kokkos. Below is relative speedup of G-PuLP on a NVidia K40 vs. PuLP on 2x IBM Power 8s (160 threads)[2]. We use common test datasets from the SuiteSparse Matrix Collection, the Stanford Large Network Dataset Collection, and a Sandia internal application mesh.



[2]White testbed at Sandia

Unfortunately ...

This straightforward parallelization strategy has some pitfalls. We note a drop in quality (edge cut) relative to our CPU version.



Why? – Note that updates are processed asynchronously during each iteration. Each thread block reads in current part assignments in one stage, then has several other processing stages before part assignments are updated. Because of GPU $\rightarrow$ thread block $\rightarrow$ thread warp scheduling, updates are often processed using "stale" part assignment data.

This isn't the first time ...

We saw this when developing XtraPuLP for distributed memory partitioning. With XtraPuLP:

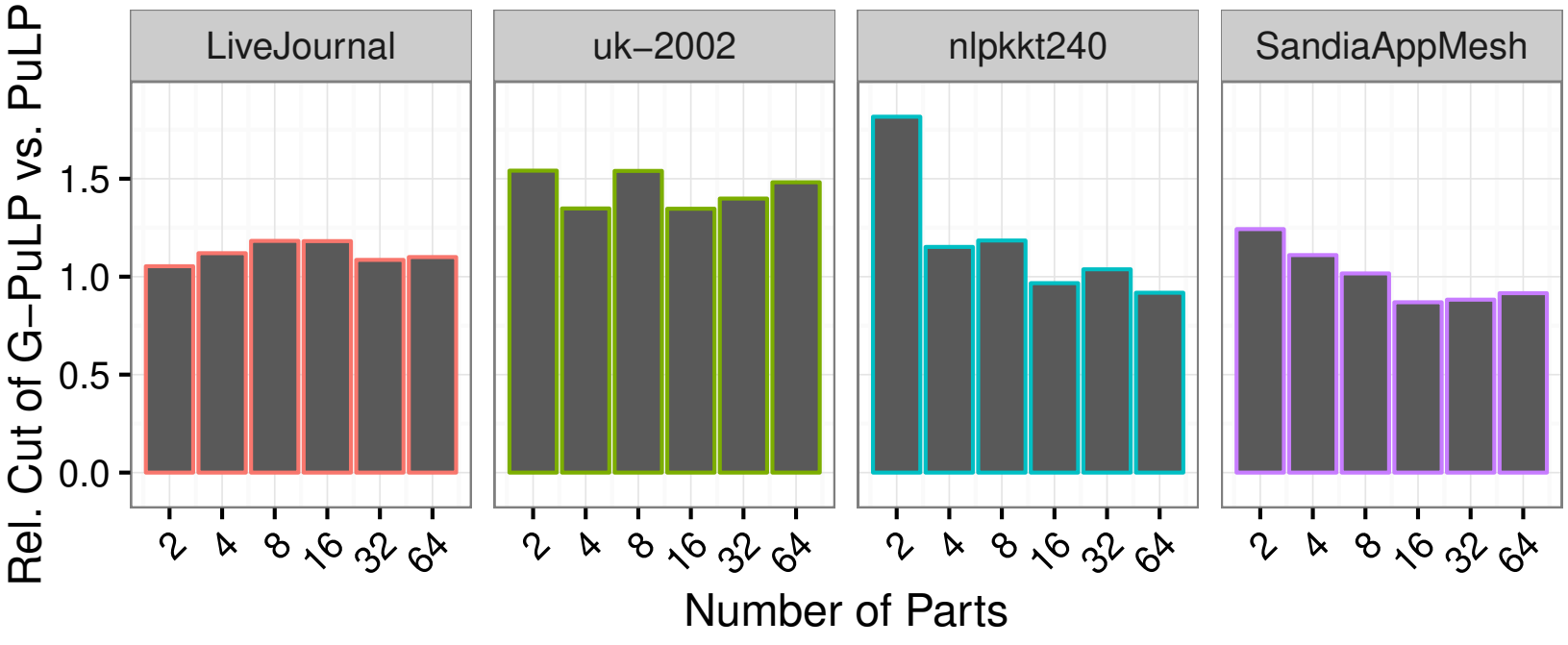
- **Observation**: wild oscillations in part assignments due to computations using stale data.
- **Solution**: slow things down, limit number of possible updates by only giving each rank a weight allocation of possible updates.

Now with G-PuLP:

- We similarly limit the amount of updates performed on a given iteration by tightening the update criteria .
- Early iterations: only updates that improve quality by at least x are allowed.
- Later iterations: we loosen this restriction by decrementing x towards zero.
- **End result**: the number of updates per-iteration decreases, which makes our asynchronous computations more "accurate".

The End Result

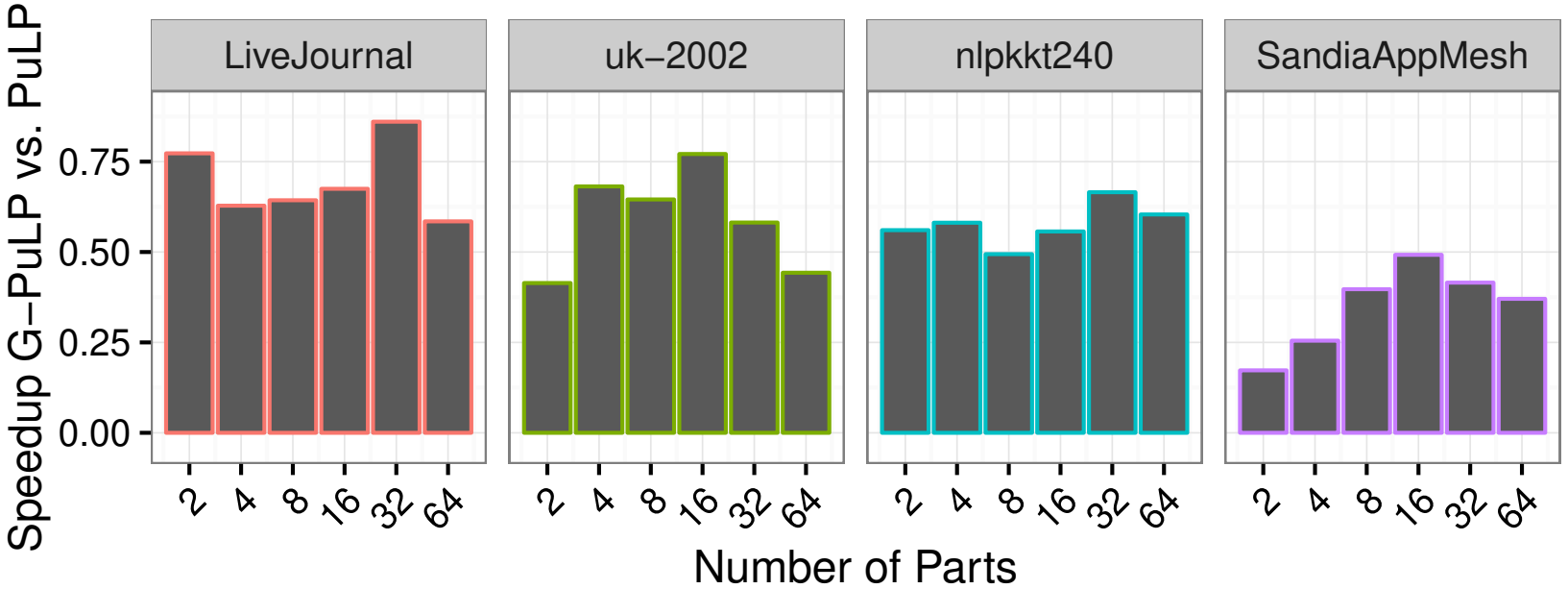
We can use this approach to close the quality gap between PuLP and G-PuLP by running G-PuLP with a larger number of iterations. Below we show relative quality using the same test setup as before.



Final Speed Results

One more "however"

However, as using our restriction for updates requires more iterations for convergence, this results in slower overall execution times for G-PuLP.



However (again), a like-for-like comparison shows that per-iteration times for G-PuLP are significantly faster (about 2× on average). This is an optimistic result, as it shows there's plenty of room to fine tune our approach in terms of the tradeoff of speed/quality.

Discussion and Future Work

We still have a bit of work to "close the gap" in terms of quality vs. speed.

- Further tune the current approach
- Investigate other parallelization schemes if current ends up being too limiting

Other future work:

- Implement in distributed memory (G-XtraPuLP)
- (reduced) Multi-level methods to improve quality while retaining speed
- Integration into Zoltan2/Trilinos

Conclusions and thanks!

Major takeaways:

- We implement shared-memory PuLP on GPU as G-PuLP.
- We can match quality or improve speed on GPU relative to CPU, but not both (yet).
- Addressing the above and scaling out to distributed memory are primary avenues of future work.

Thank you! Contact below with any questions.

slotag@rpi.edu www.gmslota.com

Bibliography I

Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008. URL `http://stacks.iop.org/1742-5468/2008/i=10/a=P10008`.

Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. I/o-efficient generation of massive graphs following the LFR benchmark. *J. Exp. Algorithmics*, 23(1):2.5:1–2.5:33, August 2018. ISSN 1084-6654. doi: 10.1145/3230743. URL `http://doi.acm.org/10.1145/3230743`.

Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E*, 78(4):1–5, October 2008. ISSN 1539-3755. doi: 10.1103/PhysRevE.78.046110.

G. M. Slota, J. Berry, S. D. Hammond, S. Olivier, C. Phillips, and S. Rajamanickam. Scalable generation of graphs for benchmarking hpc community-detection algorithms. In *SC*, 2019.