Irregular Graph Partitioning on GPUs

George M. Slota

Rensselaer Polytechnic Institute

SIAM Computer Science and Engineering 2021

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.

Welcome to the minisymposium!

Graph Partitioning for Complex Architectures and Applications

Thank you to our speakers today:

- Christopher Brisette – *Spectral clustering for compressing physical simulations*
- Abdurrahman Yasar – *Symmetric Rectilinear Matrix Partitioning for Graph Algorithms*
- Gerrett Diamond – *Diffusive Load Balancing of Particles for Distributed Unstructured Mesh Particle-In-Cell on GPUs*
- Ozan Karsavuran – *Medium-Grain Partitioning for Sparse Tensor Decomposition*

The BIG themes for this minisymposium

Graph Partitioning in General

- A long-studied, increasingly-critical preprocessing or in-situ step for many scientific and data analytic codes
- Uses: distributed load balance, graph/mesh ordering and clustering, many others

Complex Architectures

- Increasing reliance of scientific codes on GPUs
- Exascale systems: millions of threads, hierarchical memory/compute/network architectures

Complex Applications

- Physics simulations, tensor decompositions, graph and combinatorial computations

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 (particularly ones on meshes) can be represented as a graph
- 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.

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 structure or application

This talk: (Xtra)PuLP

- 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 irregular very large-scale graph inputs

Overview

Scalable Partitioning on GPUs

Big Idea: Implement (Xtra)PuLP partitioners for GPUs

- Using Kokkos for performance portability
- Currently: 1x GPU (G-PuLP); 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: GPU vs. CPU

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

GPU Processing

Why GPUs?

- 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

Considerations for XtraPuLP on GPUs

Modifying XtraPuLP for 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**
   - XtraPuLP relied on intra-node asynchronous computation for speed (and quality)
   - On GPU, this results in an *orders-of-magnitude increase* in computations on “stale” data
3. **Distributed communication**
   - Similarly, XtraPuLP originally used an asynchronous/ synchronous approach
   - This worked great for CPUs – on GPUs, the above problems are greatly exacerbated

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"

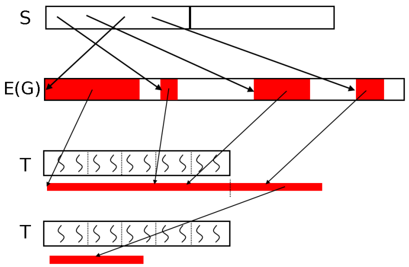
There are many proposed techniques in the literature to overcome this challenge:

- 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 for Graph-based GPU Parallelism

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

Communication - Chaos of Asynchronicity

When developing XtraPuLP's original communication strategy:

- **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.

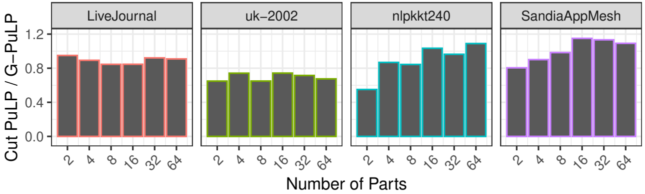
Now with G-XtraPuLP:

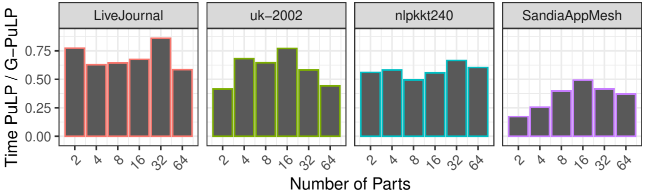
- We similarly limit the amount of updates performed by a given GPU on a given iteration by tightening the update criteria.
- Early iterations: only updates that improve quality by at least x are allowed.
- Later iterations: we loosen this restriction by decrementing x towards zero.
- **End result**: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

The tradeoffs on a single node

Single node G-PuLP vs. PuLP

Depending on how much we “limit” updates per GPU per iteration, there’s a rather large spectrum of *quality*⇔*time* for CPU and GPU PuLP. Below is possibly the best empirical case for quality, with the ratio of PuLP / G-PuLP for cut (top) and time (bottom). Bigger is better.

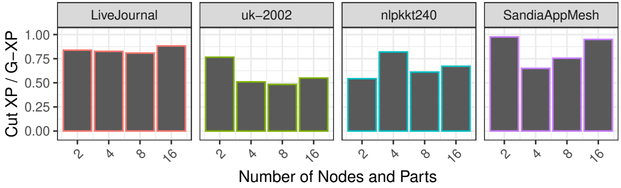


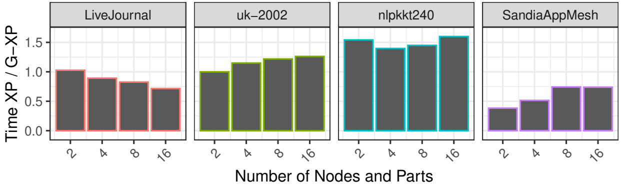


And into distributed memory with G-XtraPuLP

meh

The good: We see speedup for G-XtraPuLP vs. single node.
The bad: Time on average is worse than XtraPuLP.
The ugly: Quality can be much worse than XtraPuLP.





Number of parts computed = number of nodes run on.

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:

- (reduced) Multi-level methods to improve quality while retaining speed
- Integration into Zoltan2/Trilinos
- Hierarchical considerations for GPU/CPU or cache-based computations

Conclusions and thanks!

Major takeaways:

- We implement (Xtra)PuLP on GPU as G-(Xtra)PuLP.
- We can improve speed on GPU relative to CPU or come close in quality, 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.

Note: Looking for a PhD student to work on these problems.

slotag@rpi.edu www.gmslota.com