**Extreme Scale Parallel Graph Processing**

**Statement of Work**

**Department of Electronic Engineering and Computer Science**

**May 01, 2020**

**Updated October 14, 2020\***

**Abstract**

Data collected from different resources such as Internet, sensor network and complex systems often can be modeled as a graph, an abstract data structure preserves connectivity or relationships. Also, various problems can be formalized and represented as graphs. Because of this, graph processing is the building block to open the key between these heterogeneous types of data, allowing analysts to exploit the fundamental patterns in the data and to better understand their problems they never think about. While vast efforts have been made towards parallel graph processing, extreme large scale parallel graph processing remain challenging in many ways. Regardless of graph data format, graph processing is data centric and memory/bandwidth intensive, and it is notoriousfor irregular data access pattern and new approaches are required. Specifically, today’s large runtime systems are not designed for large scale parallel graph processing. The specific objectives of the work are to: (1) Identify the suitable data format (edge list, adjacency matrix and adjacency list) and graph primitives, then renovate a list of graph primitives along with the runtime. (2) Solve simple graph problems with synthetic data with appropriate data format and graph primitives on small scale. (3) Implement real world graph problems with graph primitives on large scale, spot the bottleneck of the processing and improve the primitives and runtime system. [[1]](#footnote-1)\*(4) Use renovated graph primitives to provide backend tools for existing Python hypergraph library to support the Hypernets project.

# Background

Data collected from different resources such as the Internet, sensor network and complex systems often can be modeled as a graph, an abstract data structure preserves connectivity or relationships. Also, various problems can be formalized and represented as graphs. Because of this, graph processing is the building block to open the key between these heterogeneous types of data, allowing analysts to exploit the fundamental patterns in the data and to better understand their problems they never think about.

In real world, the graphs representing those data can be very sparse, as the number of entities and the number of the relationships between those entities are unknown. The sparseness nature of the graph causes problems while processing parallel. There are three ways to represent graphs: edge list, adjacency matrix and adjacency list. They all have their pros and cons due to different problems and situations. But no matter which representation, graph processing is data centric and memory/bandwidth intensive, and it is famous for irregular data access pattern because of the sparseness. While vast efforts have been made towards parallel graph processing for a long time, much more is demanded. Specially, there are few large runtime systems specifically designed for extreme large scale parallel graph processing.

There are several main problems in parallel graph processing. First, a tremendous workload has been shifted to data movement, not only causing inefficient power and time consumption, but also occupying network bandwidth. Second, sparseness means the inefficient way of storing graphs among large cluster of computers, which both requires large memory to hold while processing and frequent data movement among the cluster. Third, the memory hierarchy of the standard processor becomes an architectural limit, decreasing the processing efficiency dramatically. To address these problems, the Hierarchical Identify Verify Exploit (HIVE) Program is to create a new graph analytics processor along with a runtime system to improve the processing efficiency.

Hypernets is a research project advancing the theory and application of hypergraphs as models for complex multiway data. The primary deliverable for Hypernets is an updated release of PNNL’s open sourced Python library, HyperNetX (HNX), a collection of methods and classes for the analysis of data modeled as a hypergraph. The new version will include optimized methods written in C++ and bound to a Python namespace for seamless integration into the existing framework. The optimized framework will support the practical use of HNX on very large datasets, increasing its usefulness to our sponsor. The new framework plans to exploit the capabilities developed by HIVE. \*

# Objectives

The overall goal of this study is to boost the speed of the extreme large scale parallel graph processing. The specific objectives are to:

1. Identify the suitable data format (edge list, adjacency matrix and adjacency list) and graph primitives, then renovate a list of graph primitives along with the runtime.
2. Solve simple graph problems with synthetic graph with appropriate data format and graph primitives on small scale.
3. Implement real world graph problems with graph primitives on large scale, spot the bottleneck of the processing and improve the primitives and runtime system.
4. Use renovated graph primitives to provide backend tools for existing Hypernets Python hypergraph library, HyperNetX.\*

# General Methodologies

In this project, several data formats for graphs and a list of graph primitives will be studied on standard processor (CPU, GPU). Related work from other people will become reference for the improvements on those primitives and the runtime system. Small and simple graph problems will be put to test immediately with synthetic data as it can be generated under control and behaviors more predictable. Certain assumptions will be made during this period while detailed workload characterization performed comparing with the same experiments conducted on the standard processors. Scalability will be enabled but only under small scale. Next large scale real world problems will be solved with graph primitives, along with performance characterization.

*Objective 1*: *Identify the suitable data format (edge list, adjacency matrix and adjacency list) and graph primitives, then renovate a list of graph primitives along with the runtime.*

Suitable data format for graph needs to be studied. For example, for a graph G (V, E), where

|V|=n and |E|=m, its n by n adjacency matrix implementation is easier to implement and follow. Its query time is constant, so is the addition or removal of an edge. But adding a vertex uses O(n2) time, and it consumes O(n2) space no matter how sparse the graph is. For its adjacency list implementation, it’s supper efficient to store, only O(n+m) space, constant time for vertex addition and deletion. But query time can be as worse as n. Besides the time and space cost(1), the partitioning techniques based on them are also necessary to investigate. The sparseness of the graph will demand either spatial locality(2) on single node or data movement between multiple nodes.

Because of this, different graph primitives benefit/suffer from the pros and cons of those data format implementation. Therefore, different graph algorithms implemented with different primitives must select data format carefully. All these need to be studied thoroughly by reviewing others’ work or conducting experiments.

With a clear scope of the primitives, we can proceed to renovate the graph primitives and the mechanisms of the runtime.

*Objective 2: Solve simple graph problems with synthetic graph with appropriate data format and graph primitives on small scale.*

During the graph primitives and runtime system development and after development, simple graph problems will be tackled to validate the efficiency of the design philosophy. Considering the vast amount of graph algorithms, we will mainly focus on the graph traversal related ones at this time. For example, Breadth-first search would be appropriate for this task. Breadth First Search is one of the fundamental algorithms in graph traversing or searching and has been studied more than 50 years. The first BFS algorithm was discovered by Moore(3) while trying to find the shortest paths through mazes. Lee(4) independently discovered the same algorithm in routing wires on circuit boards. It appears among many complex applications, advanced algorithms and scientific fields. It was originally used to find the shortest path in a graph but has become a building block for many other graph algorithms. Graphs are used to represent networks of communications, data organization, computational devices, etc. BFS is usually the first algorithm to perform on them if we want to understand the structure property of those graphs.

The reason to start with simple graph problems is to have a better understanding on the program behavior when it meets the new architecture design. Similar reason to use synthetic graph, it not only contributes more predictable behaviors but also the clear and well-studied data access pattern of synthetic graphs(5) (6) (7) (8). The experiment will be conducted in small scale, starting with single node, then single node with multiple non-uniform memory access (NUMA) domains, finally expanding to multiple nodes with multiple NUMA domains.

*Objective 3: Implement real world graph problems with graph primitives on large scale, spot the bottleneck of the processing and improve the primitives and runtime system.*

When the graph primitives are stable enough, real world graph problems will be implemented and put to test on large scale with the runtime system. The bottlenecks are going to be identified to help improve the primitives and runtime system. Since the we have a better understanding on BFS from the previous objective, we will proceed with graph algorithms heavily relying on BFS, label propagation algorithm for community detection(9) for instance. Label propagation algorithm is an approximation algorithm for large scale real-world community detection, uses the network structure alone as its guide. Every node is assigned a unique label in the beginning and adopts the most common labels among its neighbors at each step. As the labels propagate, densely connected nodes quickly come to a consensus to a common label, which are grouped as a community. This algorithm is based on BFS and therefore suffers the same problem BFS suffers. Since the neighbor is chosen randomly at each selection, some nodes that are on the edges of two close communities will change over iterations even if the labels of their neighbors remain constant already. Same as BFS, this is a benign race(10), which dramatically affects performance but no correctness affected.

Community detection problem is like a graph partitioning problem, which is defined as the partitioning a graph into constant groups of approximately equal sizes, minimizing the number of edges between groups. This problem is NP-hard. But label propagation method is nearly linear and therefore a perfect real-world problem to tackle with. With the previous knowledge from BFS, label propagation algorithm will enlighten us more to better graph primitives and runtime system on a larger scale.

1. *Objective 4: Use renovated graph primitives to provide backend tools for existing Python hypergraph library, HyperNetX.*\*

To further test the system on real world graphs at scale, multi-way networks modeled as hypergraphs will be represented using graph primitives and methods currently implemented in the HyperNetX and NetworkX library will be implemented and tested at scale with the runtime system. Resulting implementations will be provided to PNNL’s Hypernets project to provide an optimized backend for HyperNetX.

**Tasks:**

1. Consult with project staff on:
   1. Efficient data structures for static (immutable) hypergraphs with core functions supported by the C++ backend.
   2. PyBind module linking the C++ backend to a Python namespace.
2. Code hypergraph algorithms in C++ for static hypergraphs. Specifics TBD, examples of target algorithms include collapsing nodes and/or edges, sub-edge reduction and closure, and s-overlap, s-distance, and related hypernetwork methods like s-connected components and s-shortest paths
3. Replace current NetworkX dependencies in HNX hypernetwork science algorithms for static hypergraphs with above C++ methods. Specifics TBD, example of target algorithms include s-centralities and s-eccentricity
4. Time permitting, replace the current Python implementation for computing mod2 homology groups and Betti numbers with optimized C++ methods.

**Deliverables**

C++ code for hypergraph algorithms, s-metrics, and data structures including those methods HNX is currently importing from NetworkX.

**Level of Effort:**

20 hours per week

# Preliminary Results

From previous studies on performance characterization on shared-memory platform (SMP)/NUMA platform with many-core architectures in single node with parallel graph applications (Breadth-first Search, k-Core, Single Source Shortest Path, GraphColoring, PageRank and TriangleCount), we find that the performance of different workloads has complicated behaviors. It is heavily influenced by different architecture, different datasets, and different graph data format with even the same algorithm implementation.

From architecture aspect, a workload may take advantage of different hardware in different architecture (core, VPU, memory, cache in CPU; core, memory, device memory in GPU; memory hierarchy/topology in SMP/NUMA system), different thread placement policy, and different vectorization policy with different instruction set. Caches on shared-memory platform is to hide memory latency. For some algorithm like PageRank, based on they are computing bound or memory bound, they can gain speed up with higher bandwidth with thread oversubscription, which hides the memory latency and synchronization overhead. From dataset aspect, a workload acts differently with dataset in various structures include graph sparsity, graph topology and applications. We work with both real-world data as well as synthetic data.

Furthermore, we dug deep into Breadth-first Search algorithm. The bottom-up BFS approach can skip many edges to accelerate the search but it will not always be more efficient than the top- down approach, even though top-down always performs the worst case. Specifically, the bottom- up approach is typically more efficient when the queue is larger because it increases the probability of finding a valid parent. This leads to the direction-optimizing approach(11), a hybrid design of the top-down approach powering the search at the beginning and end, and the bottom-up approach processing the majority of the edges during only a few steps in the middle when the queue is at or near its largest.

The number of edges in the queue is used to decide when to switch from top-down to bottom-up and the number of vertices in the queue is used to decide when to switch from bottom-up back to top-down. Both the computation and the communication costs per step of top-down approach is proportional to number of edges in the queue, hence the steps when the frontier is the largest

consume most of the runtime. Conversely, the bottom-up approach is advantageous during these largest queue steps, so using the number of edges in the queue is appropriate to determine when the queue is sufficiently large to switch to the bottom-up approach. Using the heuristic as well as tuning results from prior work, once the queue encompasses at least m/10 of the edges, the bottom-up approach is likely to be beneficial. Even though the probability of finding a parent (and thus terminating early) may continue to be high as the size of the queue ramps down in later steps, there is sufficient fixed overhead for a step of the bottom-up approach to make it worthwhile to switch back to the top-down approach. Using the results from prior work, when k is the degree of the graph, we switch back to top-down approach when the vertex in the queue is less then n/14k. The degree term in the denominator ensures that higher-degree graphs switch back later to top-down since an abundance of edges will continue to help the bottom-up approach.

# References Cited

1. Chernoskutov M. Concept of parallel graph processing system for large-scale network science. In IEEE; 2017. p. 206–8. Available from: <http://ieeexplore.ieee.org/document/8109872/>
2. Linial N. Locality in Distributed Graph Algorithms. SIAM J Comput. 1992 Feb 1;21(1):193– 201.
3. Moore EF. The Shortest Path Through a Maze [Internet]. Bell Telephone System.; 1959. (Bell Telephone System. Technical publications. monograph). Available from: https://books.google.com/books?id=IVZBHAAACAAJ
4. Y. Lee C. An Algorithm for Path Connections and Its Applications. Electron Comput IRE Trans On. 1961 Oct 1;EC-10:346–65.
5. Chakrabarti D, Zhan Y, Faloutsos C. R-MAT: A Recursive Model for Graph Mining. In: Proceedings of the 2004 SIAM International Conference on Data Mining [Internet]. Society for Industrial and Applied Mathematics; 2004 [cited 2018 Mar 14]. p. 442–6. (Proceedings). Available from: https://epubs.siam.org/doi/abs/10.1137/1.9781611972740.43
6. Barabási A-L, Albert R. Emergence of Scaling in Random Networks. Science. 1999 Oct 15;286(5439):509–12.
7. ERDOS P. On random graphs. Publ Math. 1959;6:290–7.
8. de Bruijn FA, ERDOS P. On a combinatorial problem. In: Proceedings of the Section of Sciences of the Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam [Internet]. Amsterdam; [cited 2018 Mar 14]. p. 1277–9. (10; vol. 51). Available from: https://pure.tue.nl/ws/files/4300528/597479.pdf
9. Raghavan UN, Albert R, Kumara S. Near linear time algorithm to detect community structures in large-scale networks. Phys Rev E. 2007 Sep 11;76(3):036106.
10. Leiserson CE, Schardl TB. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In ACM Press; 2010 [cited 2018 Mar 15]. p.

303. Available from[: http://portal.acm.org/citation.cfm?doid=1810479.1810534](http://portal.acm.org/citation.cfm?doid=1810479.1810534)

1. Beamer S, Asanović K, Patterson D. Direction-optimizing Breadth-first Search. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis [Internet]. Los Alamitos, CA, USA: IEEE Computer Society Press; 2012. p. 12:1–12:10. (SC ’12). Available from: <http://dl.acm.org/citation.cfm?id=2388996.2389013>

1. \* Updated material introduced October 14, 2020. [↑](#footnote-ref-1)