Optimizing Parallel Graph Algorithms by Extending the GraphIt DSL

by

Tugsbayasgalan Manlaibaatar

B.S., Massachusetts Institute of Technology (2019)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author .....
Department of Electrical Engineering and Computer Science
Aug 31, 2020

Certified by .....
Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Masters of Engineering Thesis Committee

Optimizing Parallel Graph Algorithms by Extending the GraphIt DSL

by

Tugsbayasgalan Manlaibaatar

Submitted to the Department of Electrical Engineering and Computer Science
on Aug 31, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

High-performance graph processing is often very challenging because real life graphs vastly differ from each other in their sizes and structures. Therefore, we need to use many different graph specific performance optimizations and a programming system that allows domain experts to easily write high-performance graph applications.

GraphIt, a domain-specific language, is one such programming system that achieves high-performance across different algorithms, graphs, and architectures, while offering an easy-to-use high-level programming model. GraphIt decouples algorithms from performance optimizations (schedules) for graph applications to make it easy to explore a large space of optimizations. Yet, there are still many graph applications that GraphIt currently doesn't support.

In this thesis, we present a number of new additions to GraphIt to extend its' current use cases. Namely, we introduce a new operator called *intersection* that is widely used in Triangular Counting algorithm. We also introduce *functor* and *par_for* to improve current Multiple Starting Point applications by adding nested parallelization. Using the new features, we are able to get up to 16x speedup over the GraphIt implementation without the added features on road graphs that don't benefit from single level parallelization.

Thesis Supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to express my gratitude to many people without whom this thesis would not have been possible.

I would like to thank my thesis advisor, Professor Saman Amarasinghe, for patiently guiding me through my MEng studies. Ever since I started working in his research group, he has always been very helpful and always reminded me the big picture of the research and high-level future directions. I also want to thank Professor Julian Shun for giving valuable feedbacks throughout my SuperUROP and MEng studies.

I would also like to acknowledge my mentor, Yunming Zhang, for being there for me all the time. I had the pleasure of working with him ever since Fall 2018 on various projects for my SuperUROP and MEng. He was extremely hands-on and helpful for everything I did and taught me many techniques in performance engineering and research. He also spent a lot of time helping me to debug my code and even gave me useful tips for job search and technical interviews.

I want to thank Ajay Brahmakshatriya for his help on brainstorming and many C++ crash courses. He was always very responsive to my questions and helped me tremendously through my MEng studies.

Next, I would like to express my appreciation for other members of the COMMIT group. I am thankful to Changwang Hong for joining our meetings and providing useful insights; to all the other members for the great research talks during our Friday group meetings.

I want to thank my parents, Oyunbeleg Bayandorj and Manlaibaatar Choidogsuren, for raising me and providing me with the opportunity to study in the US. I am also grateful for my siblings and my grandparents for their love and support. I would not be here without the strong support from my family members.

Finally, I want to thank my Buddhist community back in Mongolia for their prays and wishes. I want to thank Altangerel Zundui for his valuable guidance.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

Graph analytics is widely used in many large-scale real-world applications. For example, friend suggestion problem can be modeled as triangle counting on a social graph [9], fraud detection is same as detecting a cycle in a transaction graph, and subgraph matching algorithms can be used for detecting cancer in a protein interaction network [17]. Many real-world graphs, however, are large in size and irregular in shape, thus imposing a challenge to many text book graph algorithms. Therefore, we want to achieve high performance graph algorithms by considering both theoretical improvements and real-life properties of graphs.

1.1 Motivation

GraphIt [26, 27] is a new domain-specific language that achieves high-performance across different algorithms, graphs, and architectures, while offering an easy-to-use high-level programming model. GraphIt decouples algorithms from performance optimizations (schedules) for graph applications to make it easy to explore a large space of cache, NUMA, load balance, and data layout optimizations. As of now, GraphIt supports a wide range of applications, many different data structures, and numerous performance optimizations.

However, there are still a number of applications that GraphIt doesn't support yet. For example, GraphIt originally doesn't support Triangular Counting Algorithm

which is specified as one of the applications in GAP Benchmarking Suite [4]. GAP is used for evaluating high performance graph algorithm frameworks by providing benchmarking specifications for five algorithms and six different graphs. (total of 30 tests). GAP also provides high-performance reference implementations for those algorithms.

Furthermore, some applications in GraphIt can be improved even further. For instance, our initial benchmarks reveal that the current GraphIt centrality measure applications that are run from multiple starting points can be further improved by running those starting points in parallel. Therefore, the goal of this thesis is to extend GraphIt's compiler to support Triangular Counting and faster multiple starting point applications.

1.2 Contribution

In this thesis, we present a number of improvements to the ongoing development of GraphIt. We can frame the improvements into three main categories.

Firstly, we implement Triangular Counting Algorithm in GraphIt in accordance to GAP Benchmarking suite. The key idea of the implementation is that Triangular Counting algorithm can be entirely written as the intersection of neighboring sets for each edge. Since each vertex can have a different characteristic requiring us to use different intersection methods, we use extended GraphIt to schedule different types of intersections. Using different schedules, we are able to obtain faster implementation than the Triangular Counting algorithm in GAP which uses the naive intersection algorithm. We discuss our optimization more in detail in Chapter 3.1.1.

Secondly, we implement a set of centrality measure applications and a simple local graph clustering algorithm that can be run from multiple starting points. Even though GraphIt already has the versions of those applications where each starting points are run serially (there is a parallelism within the starting point, but not across all starting points), we want to achieve better performance by parallelizing over the starting points by introducing new programming models into GraphIt. Specifically,

we introduce configurable *parallel for* to parallelize over starting points and *functor* to operate on starting point specific data structures. We discuss these new additions more in detail in Chapter 4 and 5. By using these features, we are able to get up to 16x speedup over the GraphIt implementation with serial starting points on certain graphs.

Finally, we develop several optimizations over the naive implementation on an approximate maximum inner product search algorithm which is used as a part of the annual evaluation from DARPA SDH program [1]. This application is an interesting use case of calling external functions in GraphIt where the data parsing part is done in python and some part of the search algorithm is written as external C++ functions. The approximate algorithm involves three phases which can be optimized independently:

1. Graph Setup Phase: We introduce a faster graph builder for GraphIt using parallel prefix sum.

2. Greedy Walk Phase: We optimize this phase by rewriting it in C++ from python.

3. Query Search Phase: Since each query is almost independent from each other, we parallelize over multiple queries. We get great speedup since the work done for each query is very small so that we don't have to parallelize within each query. By not parallelizing within each query, we can easily spawn many threads when parallelizing over multiple query points.

Using above optimization strategies, we get nearly 7x speedup over the naive implementation. We discuss the optimization strategies more in detail in Chapter 3.3.

1.3 Thesis Organization

In Chapter 2, we give a brief overview on graph algorithms that are used in this thesis and talk about potential optimization ideas. We also talk about the infrastructure of

the GraphIt compiler. Significant portion of this thesis serves as an extension to the GraphIt compiler. Finally, we talk about our initial benchmark results for Multiple Starting Point applications.

In Chapter 3, we present some of the performance optimization steps we took while working on different graph applications and some more general optimizations that were helpful across the framework. In Chapter 4, we present the design of newly added features in GraphIt. In Chapter 5, we discuss the important implementation details of the newly added features from the compiler point of view. In Chapter 6, we present some use cases to show how all the new features fit together.

In Chapter 7, we analyze our new optimizations with respect to the GraphIt version without these optimizations. Finally, Chapter 8 summarizes our work and talks about some future research directions.

Chapter 2

Background

In this chapter, we talk about notations we use throughout this thesis and different graph algorithms we use for evaluation. Then, we give a brief overview on GraphIt compiler which we extend for our use cases. Finally, we give a background on Multiple Starting Point applications and talk about initial benchmarks to motivate the problem.

2.1 Notations

We denote a graph network by $G = (V, E)$ where V represents the set of vertices and E represents the set of directed edges. A single vertex is denoted by v. We also denote $|V|$and $|E|$ as the number of vertices and the number of edges respectively.

We denote $G.nghs(v)$ for a vertex v as a list of neighbors of v ordered by their IDs. For example, if a vertex v is a neighbor of 3, 5, 7, and 4, then the output will be $\{3, 4, 5, 7\}$. We also denote $G.deg(v)$ for the degree of a vertex v. In addition, DAG refers to a directed acyclic graph.

2.2 Graph Algorithm Descriptions

In this section, we talk about different graph algorithms used for this thesis. We use Triangular Counting as an example of *intersection* operator. In addition, we

use several centrality measures and local graph clustering algorithm as use cases for our new operators *functor* and *parallel for*. These applications are chosen as they can be from multiple sources that can be parallelizable. Finally, we briefly discuss about IPNSW algorithm which is used for approximating maximum inner product search. Finally, we discuss about our approach to implementing Multiple Starting Point applications including initial benchmarks.

2.2.1 Triangular Counting

Triangular Counting counts the number of triangles (cliques of size 3) in the graph. It counts each triangle once regardless of the permutation of its constituent vertex identifiers. The most of the computation breaks down into intersecting two sorted sets of potentially very different sizes. Below we present the pseudocode for Triangular Counting algorithm:

Algorithm 1: Triangular Counting Algorithm

```
   Data: E
   Result: count
1  count ← 0;
2  for v ∈ V do
3  |  v_degree ← G.degree(v);
4  |  for u, w ∈ N(v) do
5  |  |  if G.degree(u) > v_degree && G.degree(w) > v_degree then
6  |  |  |  if can_form_triange(u, v, w) then
7  |  |  |  |  count ← count + 1;
8  |  |  |  end
9  |  |  end
10 |  end
11 end
```

Line 5 ensures that we do not count a same triangle triplets multiple times by considering only vertices with higher number of neighbors.

Many frameworks such as GraphIt and Ligra [23] work by providing user-defined functions over vertices and edges. Since we aim to develop a fast Triangular Counting Algorithm on GraphIt, it is essential to frame this algorithm as an operation over edges or vertices. Below pseudocode shows the one possible implementation:

Algorithm 2: Triangular Counting over edges

```
   Data: E
   Result: count
1  count ← 0;
2  for (src, dest) ∈ E do
3  |   count += intersect(G.nghs(src), G.nghs(dest))
4  end
```

We can pass in *dest* to the intersection method in Line 3 to make sure we don't count vertices that have higher ID than *dest*. This will eliminate the issue of over counting triangles.

We can see that Triangular Counting is all about intersecting neighboring vertices of the *src* and *dest* of every edge.

2.2.2 Betweenness Centrality

Betweenness Centrality (BC) of the vertex v measures the importance of the vertex in a graph network. Given a graph $G = (V, E)$ and vertices $s, t \in V$, let σ_{st} be the number of the shortest paths from s to t in G, and $\sigma_{st}(v)$ be the number of the shortest paths that pass through a specified vertex v. Then, BC of a vertex v: $BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$. Brandes algorithm [6] efficiently computes BC in $O(|V| \times |E|)$ time on unweighted graphs based on a new accumulation technique. By defining the *dependency* of a source vertex s on any given vertex v as: $\delta_s(v) = \sum_{t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$, we can rewrite a BC of v: $\sum_{s \neq v \in V} \delta_s(v)$. Given pairwise distances and shortest path counts, we can also see the following recursive relation

$$\delta_s(v) = \sum_{w : v \in P(w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_s(w))$$

where $P(w)$ is the set of immediate predecessors of vertex w on the shortest paths starting from s [24]. Brandes algorithms works by accumulating dependencies using above relation. It performs breadth-first-search (BFS) (forward pass) to count the number of shortest paths and constructs DAG of the shortest paths from each vertex. Then, the algorithm traverses the DAG (backward pass) to accumulate dependen-

cies and add them to BC score. Pseudocode below shows the high level Brandes Algorithm.

Algorithm 3: Brandes Algorithm

```
   Data: G
   Result: BC
    // Initialize BC score array
1  forall v ∈ V do
2  |  clear BC[v]
3  end
4  foreach v ∈ V do
   |   // Initialize
5  |  forall v ∈ V do
6  |  |  clear σ_st v, P(v), δ_s(v)
7  |  end
   |   // Construct shortest path DAG (forward pass)
8  |  forall v ∈ V do
9  |  |  compute σ_st v, P(w) using BFS
10 |  end
   |   // Backward pass to compute BC scores
11 |  forall v ∈ DAG do
12 |  |  compute δ_s(v)
13 |  |  BC[v] += δ_s(v)
14 |  end
15 end
```

Computing exact score takes unreasonably long time as it essentially requires us to run Brandes algorithm [6] from every single vertex in the graph G as can be seen from Algorithm 3. Therefore, we only use few starting vertices to approximate the total score. In this thesis, we aim to get faster performance by parallelizing over those starting vertices.

2.2.3 Closeness Centrality

Closeness Centrality (CC) of the vertex v measures the importance of the vertex in a graph network and it is based on the ensemble of its' distances to all the other nodes in the graph. More formally, we define the CC score of a vertex v as:

$$CC(v) = \frac{1}{\sum_{t \in V} d(v,t)}$$

where $d(v,t)$ is a shortest distance between vertices t and v. If the CC metric of this vertex v is high, it implies that the node is important as this vertex is relatively close to other vertices. One of the typical ways to calculate CC scores for unweighted graphs is to run BFS from the vertex of interest, and store the distances. In the cases of weighted graphs, we make use of Dijkstra Algorithm to compute the shortest distances. Algorithm 4 shows the pseudocode for Closeness Centrality on undirected graphs using BFS.

Algorithm 4: Closeness Centrality Algorithm for undirected graphs

```
   Data: G
   Result: BC
    // Initialize CC score array
 1 forall v ∈ V do
 2 |   clear CC[v]
 3 end
 4 foreach v ∈ V do
 5 |   forall v ∈ V do
 6 |   |   clear v
 7 |   end
 8 |   forall w ∈ V do
 9 |   |   compute depth(w) using BFS
10 |   end
11 |   CC[v] = 1/depth.sum()
12 end
```

2.2.4 Local Graph Clustering

Graph Clustering has many important applications in computing, but due to the increasing size of the real life graphs, it is computationally expensive to find the global graph clustering as you have to look at the entire graph in most cases. The local graph clustering algorithms mitigate this issue by just returning a single cluster based on the seed node. These algorithms scale well [11] in practice because they only depend on the size of the cluster rather than the entire graph. In this thesis, we extend one of local clustering algorithms, PageRank-Nibble algorithm [2], to support multiple seeds. PageRank-Nibble works by iteratively spreading the probability mass

around graph. Below we present the pseudocode for Pagerank-Nibble:

Algorithm 5: Pagerank-Nibble Algorithm

```
1  r ← {(seed, 1.0)}
2  p ← {}
3  while any vertex u satisfies r[u]/G.deg(u) ≥ ϵ do
4      Choose any vertex u where r[u]/G.deg(u) ≥ ϵ
5      p[u] ← p[u] + αr[u]
6      for ngh ∈ G.nghs(u) do
7          r[ngh] ← r[ngh] + (1 − α) r[u] / (2×G.deg(u))
8      end
9      r[u] = (1 − α) r[u] / 2
10 end
```

Note that the seed initially has a probability of 1.0 (Line 1)and iteratively spreads it to other vertices.

2.2.5 Maximum Inner Product Search

Maximum Inner Product Search (MIPS) is important in a number of machine learning tasks such as efficient Bayesian inference, memory networks training, and reinforcement learning. MIPS essentially tries to find a set of points that have the maximum dot product with a query point. Given a set of d dimensional vectors X and a query vector q, the goal of MIPS is to find the K elements of X that have the largest inner product with q. The inner product is a commonly used measure of similarity between two vectors. Thus, being able to find elements that have a large inner product with your query is useful for things like search engines or recommender systems. As a reminder, for a vector $a = [a_1, a_2, \ldots, a_d]$ and a vector $b = [b_1, b_2, \ldots, b_n]$, we define the dot product as:

$$a * b = \sum_{i=1}^{d} a_i b_i$$

The brute force approach would be to compute the inner product of all vectors in X with the query point and choose the top K. This has runtime of $O(|X| * d)$ and as the number of vectors increase, this approach gets computationally slow and expensive.

Hence, algorithms like Inner Product Navigable Small World (IPNSW) [19, 18] try to reduce the cost of each query by building a special data structure called an

"index" that can be used to do cheap, fast and approximate MIPS. Further details will be discussed in Chapter 3.3.

2.3 GraphIt Overview

This thesis contributes to the ongoing development of the GraphIt, a high performance DSL for graph analytics. GraphIt achieves consistent high-performance across different algorithms, graphs, and architectures while offering an easy-to-use high-level programming model. GraphIt achieves this by decoupling the algorithm specification from optimization strategies for graph applications. Many graph applications require different optimization techniques therefore users normally have to try out a large set of such techniques to achieve high performance. GraphIt solves this problem by separating the high-level algorithms from performance optimizations.

Users specify graph algorithms using the algorithmic language involving just high-level operations on sets of vertices and edges. They use the separate scheduling language to compose different optimizations. The algorithmic language exposes different high-level optimization opportunities, such as parallelization and edge traversal direction.

The scheduling language supports a large space of optimization techniques, such as edge traversal direction, data layout, parallelization, cache efficiency, NUMA, and kernel fusion optimizations. GraphIt uses scoped labels to target specific operations to optimize. Moreover, it uses an abstract *graph iteration space* model to represent, compose, and ensure the correctness of edge traversal optimizations.

GraphIt compiler has three main components: a frontend that scans and parses the high-level algorithm and the specific optimization chosen by the user, a midend that interprets and analyzes through schedules via multiple lowering passes to ensure correctness, and a backend that generates high-performance C++ implementations with given optimizations. This section gives a brief overview on how each of these components work together.

2.3.1 GraphIt Frontend

GraphIt frontend is adapted from the frontend of Simit [13], a DSL for physical simulation, to handle scanning, tokenizing, parsing, and semantic analysis. The frontend supports various data types such as primitive types, edgeset, vertexset, and many more. In addition, the frontend has many built-in operators including parallel sum, parallel max/min over arrays and other atomic operators.

GraphIt's scheduling language exposes a family of C-like function calls with common ***config*** prefix followed by the name of a specific optimization and additional function arguments. For instance, ***configApplyParallelization*** can be used to specify what kind of parallelization strategy users want to use and ***configApplyDirection*** is used for traversal direction optimizations. Figure 2-1 shows some of the supported schedules in GraphIt.

Apply Scheduling Functions	**Descriptions**
`program->configApplyDirection(label, config);`	Config options: **SparsePush**, DensePush, DensePull, DensePull-SparsePush, DensePush-SparsePush
`program->configApplyParallelization(label, config, [grainSize], [direction]);`	Config options: **serial**, dynamic-vertex-parallel, static-vertex-parallel, edge-aware-dynamic-vertex-parallel, edge-parallel
`program->configApplyDenseVertexSet(label, config, [vertexset], [direction])`	Vertexset options: **both**, src-vertexset, dst-vertexset Config Options: **bool-array**, bitvector
`program->configApplyNumSSG(label, config, numSegments, [direction]);`	Config options: **fixed-vertex-count** or edge-aware-vertex-count
`program->configApplyNUMA(label, config, [direction]);`	Config options: **serial**, static-parallel, dynamic-parallel
`program->fuseFields({vect1, vect2, ...})`	Fuses multiple arrays into a single array of structs.
`program->fuseForLoop(label1, la bel2, fused_label)`	Fuses together multiple loops.
`program->fuseApplyFunctions(label1, label2, fused_func)`	Fuses together two edgeset apply operators. The fused apply operator replaces the first operator.

Figure 2-1: Some GraphIt Schedules [26]. The default option for an operator is shown in bold. Optional arguments are shown in [ ]. If the optional direction argument is not specified, the configuration is applied to all relevant directions. We use a default grain size of 256 for parallelization

The frontend parses the high-level algorithm code to construct frontend Intermediate Representation (FIR) objects to represent the program. Moreover, the frontend reads scheduling C-like function calls and constructs scheduling objects that contain three types ofinformation: the physical layout of the vertex data, the edge traversal optimization, and the frontier data structures. The frontend then passes the FIR

objects along with the scheduling objects to the midend for further analysis.

2.3.2 GraphIt Midend

The midend consists of multiple optimization and correctness lowering passes which transform the scheduling objects and FIRs into a midend Intermediate Representation (MIR) objects that are used for backend code generation.

In general, the midend lowering passes are responsible for gathering any global information concerning this pass from the MIR context and modifying local properties of MIR objects. For instance, through midend lowering passes, GraphIt inserts atomic operators whenever certain data structures can potentially be accessed by multiple threads. We refer to this pass as *dependency analysis* pass. Just like FIR, MIR nodes are abstract structures that represent things like expressions, statements, and variable declarations. For example, ***EdgeSetApplyExpr*** is an MIR object that contain information such as whether the edge traversal should enable vertex deduplication. GraphIt heavily uses a visitor pattern, where an additional visitor class implements the appropriate specializations in each MIR node. Each of the lowering passes constructs an MIR visitor or rewriter, gets a list of function from the MIR context, and "visits" each function to modify MIR nodes according to the scheduling objects.

2.3.3 GraphIt Backend

After the midend finishes processing the program, MIR nodes are recursively visited for the code generation. For instance, if the *is_parallel* flag of ***EdgeSetApplyExpr*** is set to true, the backend generates C++ code with parallel primitives such as OPENMP, CILK, or TBB.

2.4 Multiple Starting Point Applications

Many algorithms like Betweenness Centrality (BC), Closeness Centrality (CC), and PR-Nibble are usually run from a single starting vertex to compute scores for each vertices. Therefore, there are many fast algorithms that efficiently parallelize the computation within each starting point. For example, GraphIt uses a direction-optimized BFS [3] that achieve high parallelization on multicore machines by switching between top-down and bottom-up approaches depending on the frontier size for the forward run of BC and unweighted version of CC.

But, it is sometimes necessary to run above applications from multiple starting points. For example, in BC, the exact score for each vertex is calculated by running BC from every vertices in the graph. Since it is computationally too expensive, we usually approximate BC score by running from certain chosen vertices. For instance, GAP Benchmarking Suite [4] use 4 vertices. In the case of CC, running from a single source only gives you a score for the source. Therefore, we have to run CC for every vertices we are interested in.

While it is certainly possible to run those chosen starting points sequentially, we want to further utilize hardware by running the starting points in parallel. However, we can't run the starting points completely independent because all the computations done within each starting points have to converge back to a global answer in the end.

2.4.1 Initial Benchmarks

As per our discussion above, we hope to gain better performance on multiple starting point applications by further parallelizing (parallelizing over starting points). However, we have to ensure following two properties through our initial benchmarks.

- **Parallelization within each starting point**: If the parallelization within each starting point is already too high, we won't get any performance gain by parallelizing over starting points. For example, if the program scales close to 24 times on a 24-core machine relative to its' serial version, it implies that the program almost fully takes advantage of the hardware. Therefore, parallelizing

over starting points in this case wouldn't help. On the other hand, if the program scales many fewer than 24 times, we might be able to get more scalability by parallelizing over starting points.

- **Allocation of local data structures**: For many multiple starting point applications, we need to maintain local arrays to keep track of the local score from each starting points which, in the end, are reduced to global score array. Therefore, we need to make sure that allocating such local arrays don't take too much of the computation time.

Scalability Within A Single Starting Point

In this experiment, we run Betweenness Centrality on 10 different starting points and average the results for both one core and 24 cores. Table 2.1 summarizes the result for five graphs. The descriptions of the graphs can be found in section 7.2. As you can see from the table, Road-USA has a very bad scalability (around 1.4x) while Kron and Twitter have good scalabilities (around 16x - 18x). In addition, socLive and Web graphs have some mild speedup of around 12x. Therefore, we can at least expect to get better performance by parallelizing over starting points on graphs such as socLive, Web, and Road-USA.

Graphs	One Core, s	24 Cores, s	Scalability
Web	16.512	1.347	12.26
Road-USA	4.717	3.388	1.39
Kron	77.784	4.27	18.22
Twitter	28.118	1.74	16.16
socLive	1.4838	0.117	12.70

Table 2.1: Average scalability within each starting point of Betweenness Centrality.

Local Array Allocation

In this experiment, we want to show that allocating local arrays within each starting point is not the bottleneck of the performance. For that, we run Betweenness

Centrality algorithm on 10 different starting points and average the results. As you can see from Table 2.2, it is clear that allocating local arrays and initializing local arrays is very small compared to the main body of the program. Note that the time of initializing local arrays consists of allocating the necessary memory and clearing them. For example, we only spend 0.231 seconds on initializing arrays while we spend 3.361 seconds on the main computation on Kron graph. Reducing the local score to the global score takes even less time than the local array initialization. Therefore, we can conclude that we can safely use local arrays for each starting point when running multiple starting point applications. We, however, have to deallocate local arrays frequently to prevent running out of memory since some of the graphs are very big. It is also

Graphs	Local Array Allocations, s	Main Body, s	Reduce, s	Total, s
Web	0.103	1.378	0.017	1.490
Road-USA	0.058	3.557	0.008	3.623
Kron	0.231	3.361	0.046	3.638
Twitter	0.123	1.617	0.021	1.761
socLive	0.025	0.088	0.002	0.115

Table 2.2: Average Breakdown of Total Computation Time for Betweenness Centrality. Local Array Allocation column refers to the time it requires to initialize local arrays necessary for BC. The Main Body column refers to the forward and backward passes of BC. The Reduce column refers to the time that requires to reduce the local score to the global score.

2.5 Summary

In this chapter, we gave a brief overview of GraphIt compiler which we will extend further for Triangular Counting algorithm and Multiple Starting Point Applications. We also conducted some initial experiments to motivate our work in the future sections regarding Multiple Starting Point applications. In addition, we gave detailed descriptions of graph algorithms we use for the evaluation of this thesis.

Chapter 3

Performance Optimizations

In this section, we present compiler optimizations we develop for Triangular Counting Algorithm and Multiple Starting Point applications, and handwritten optimizations for IPNSW applications. Since the optimizations for IPNSW are hard to follow without extensive background information, we also present how IPNSW works in detail. Firstly, we talk about different intersection methods we can use for optimizing Triangular Counting Algorithm as GraphIt schedules. Then, we talk about compiler optimizations we develop for direction-optimized versions of Multiple Starting Point applications. Finally, we talk about our hand-optimization version of IPNSW.

3.1 Optimizations for Triangular Counting Algorithm

Intersecting two sorted sets is an essential part of Triangular Counting Algorithm. As we discussed in Chapter 2.2.1, Triangular Counting boils down to many sorted set intersections with different sizes because some vertices can have few neighbors while others can have a large number of neighbors in power-law graphs. Therefore, a method that works well with particular combination of two sorted sets might not work well with other settings. As a result, we need to have multiple intersection methods that can be switched easily. Therefore, we aim to optimize the baseline Triangular Counting in GAP which uses only one type of intersection.

3.1.1 Different Schedules

We support following set intersection methods with the *intersection* operator.

- **NaiveIntersection**: A standard way to compute the intersection where we have two pointers that are incremented one by one. Algorithm 6 shows the pseudocode for the naive intersection. It is also the intersection method used in GAP Benchmarking Suite.

Algorithm 6: Naive Set Intersection Method

```
   Data: A, B
   Result: count
 1 pointer_A ← 0
 2 pointer_B ← 0
 3 count ← 0
 4 while pointer_A < A.size() && pointer_B < B.size() do
 5 |   if A[pointer_A] > B[pointer_B] then
 6 |   |   pointer_B += 1
 7 |   else if A[pointer_A] < B[pointer_B] then
 8 |   |   pointer_A += 1
 9 |   else
10 |   |   pointer_A += 1
11 |   |   pointer_B += 1
12 |   |   count += 1
13 |   end
14 end
```

- **HiroshiIntersection**: Compare elements of two sets 3 by 3. This method was inspired from the paper by Inoue et al. [12]. For the sake of simplicity, Algorithm 7 shows the pseudo code for 2 by 2 intersection. This type of intersection significantly reduces branch misprediction overhead by replacing few expensive (e.g. inequality check) branch mispredictions with simpler branch mispredictions (e.g. equality check). For instance, in Algorithm 7, we compare $2 \times 2 = 4$ pairs for every two elements as opposed to 2 pairs in the naive set intersection algorithm and Line 18 shows the hard-to-predict conditional branch (highlighted in red). Even though this approach does more work in general, it reduces the branch misprediction cost significantly. Therefore, we get overall performance improvement when intersecting two sets with **comparable** sizes.

- **BinarySearchIntersection**: This method intersects two sets by looking up the elements of the smaller set in the larger set in a binary search fashion.

Algorithm 7: Hiroshi Intersection Method

```
   Data: A, B
   Result: count
1  pointer_A ← 0
2  pointer_B ← 0
3  count ← 0
4  while (1) do
5  |   A_0 = A[pointer_A]
6  |   A_1 = A[pointer_A + 1]
7  |   B_0 = B[pointer_B]
8  |   B_1 = B[pointer_B + 1]
9  |   if A_0 == B_0 then
10 |   |   count += 1
11 |   else if A_0 == B_1 then
12 |   |   count += 1
13 |   |   goto advanceB
14 |   else if A_1 == B_0 then
15 |   |   count += 1
16 |   |   goto advanceA
17 |   end
18 |   if A_1 == B_1 then
19 |   |   count += 1
20 |   |   goto advanceAB
21 |   else if A_0 > B_1 then
22 |   |   goto advanceB
23 |   else
24 |   |   goto advanceA
25 |   end
26 |   advanceA:
27 |   pointer_A += 2
28 |   if pointer_A ≥ Aend then
29 |   |   break
30 |   else
31 |   |   continue
32 |   end
33 |   advanceB:
34 |   pointer_B += 2
35 |   if pointer_B ≥ Bend then
36 |   |   break
37 |   else
38 |   |   continue
39 |   end
40 |   advanceAB:
41 |   pointer_A += 2; pointer_B += 2
42 |   if pointer_A ≥ Aend || pointer_B ≥ Bend then
43 |   |   break
44 |   end
45 end
   // fall back to naive intersection
```

Since we assume that each set is sorted, the search space reduces each time we look up an element. For example, if we find the match of the first element of the smaller set at index 10 of the bigger set, we can start the search of the second element of the smaller set starting at index 10 of the bigger set. This strategy works well for the cases where input sets are extremely unbalanced. The pseudocode is given in Algorithm 8.

Algorithm 8: Binary Search Intersection Method

```
   Data: A, B
   Result: count
 1 start ← 0
 2 prevStart ← 0
 3 count ← 0
 4 for i ← 0 to A.size() do
       // binarySearch method looks up A[i] in set B at starting index of start
 5     start ← binarySearch(A, B, i, start)
 6     if start ≠ −1 then
 7         prevStart ← start
 8         count += 1
 9     else
10         start ← prevStart
11     end
12 end
```

- **MultiskipIntersection**: This method is similar to **NaiveIntersection** but it increments the pointer by some k. For the correctness purposes, whenever the pointer skips through a potential intersection, it checks surrounding window of size k. Empirically, we find that $k = 3$ gives the best result thus 3 is used internally. Even though this method does roughly the same amount of work as naive intersection method in the worst case (e.g. intersection is dense), we observe some speedup in real life cases because intersection is sparse. Algorithm 9 shows the pseudocode for this approach.

3.2 Optimizations for Multiple Starting Points

3.2.1 Vertex Deduplication for Multiple Starting Points

Vertex Deduplication is used to make sure that each vertex is inserted into the outgoing edges only once. In general, we maintain a deduplication flag array which is

Algorithm 9: MultiSkip Intersection Method

```
   Data: A, B
   Result: count
 1 pointer_A ← 0
 2 count ← 0
 3 for i ← 0 to B.size() do
 4 |   B_stop ← B[i]
 5 |   while pointer_A < A.size() && A[pointer_A] < B_stop do
 6 |   |   pointer_A += 3
 7 |   end
 8 |   if A[pointer_A] == B_stop then
 9 |   |   count += 1
10 |   else
   |   |    // If above is not true, we rollback by 2 to make sure we do not skip possible
   |   |     intersection
11 |   |   pointer_A −= 2
12 |   |   if A[pointer_A] == B_stop then  count += 1
13 |   |   pointer_A += 1
14 |   |   if A[pointer_A] == B_stop then  count += 1
15 |   |   pointer_A += 1
16 |   end
17 end
```

an array of integers that can be either 0 or 1 to keep track if the vertices are already included. Therefore, the size of the array is equal to the number of vertices in the graph. Depending on traversal direction, we might need to add compare-and-swap (CAS) on the deduplication flag array for correctness. We use integers instead of booleans for the deduplication flag array because CAS is not supported on booleans. Since applications such as Betweenness Centrality and Closeness Centrality are direction-optimized, we need to use vertex deduplication with CAS as they include both traversal directions. Originally, GraphIt uses a single deduplication flag array which is a field of the input graph object but this is not thread safe if we run multiple starting points in parallel.

Hence, we modify GraphIt compiler to maintain a simple memory manager of local deduplication flag arrays. Then, whenever a thread needs a deduplication flag array, it gets the latest available flag array from the memory manager. In addition, a thread returns the deduplication flag back to the memory manager as soon as it is done. By doing so, we guarantee that the number of deduplication flag arrays is at most equal to the number of outer threads used in parallelizing over starting points. It is fine for other threads to work on already used deduplication flag array since all we care is if the value at certain indices are flipped during the process. Each operation

(asking for a flag and returning a flag to the pool of flags) to the memory manager is guarded by a global lock to ensure correctness.

3.3 Optimizations for IPNSW

As we discussed in Chapter 2.2.5, we only focus on "search" portion of the IPNSW algorithm (Figure 3-1). When we do a "search", we "zoom in" from the broadest similarity graph to the finest. Roughly, given a query q, we interpret as "find the large cluster that q is close to, then find a subcluster that q is even closer to, then find a subcluster ...". A good analogy would be if we are trying to efficiently find all of person q's family members, we might start by:

1. find everyone from same city as q, then
2. find everyone from same district as q, then
3. find everyone from same street as q ...

Therefore, this procedure eliminates the search space significantly. To implement this procedure, we need to do the following:

1. **Constructing hierarchical graphs from the input**. In our analogy, the first level would be the city, the second would be the district, and so on.
2. **Greedy walk to find the most optimal subgraph to start**. We start from a "entrypoint node" at the highest level (city in our analogy). At every step, we "greedily" walk to the neighbor of the current node with the maximum similarity to our query q. When we reach the local maximum where there is no neighbors that has higher similarity than the current node, we "descend" to the next "zoomed in" graph and continue the walk. (descending from a city level to district level in our analogy)
3. **Find the closest points to multiple queries** Once we enter the appropriate level, we do "beam search" procedure as described in Figure 3-1.

```
// Global counter for number of distance computations
global NUM_DIST_COMPUTATIONS = 0

function dist(Node a, Node b)
	global NUM_DIST_COMPUTATIONS += 1
	return -1 * (dot product of a and b's features)

function searchKNN(Graph GO, List[Graph] Gs, Node q, Node v_entry, Int ef, Int
n_results)

	// Initialization
	v_curr = v_entry
	results = {} // Empty set
	candidates = {} // Empty set
	d_worst = Infinity

	// Greedy Walk Phase
	for G in Gs:
		while not converged:
			neighbors = nodes adjacent to v_curr in G
			for neib in neighbors:
				if dist(neib, q) < dist(v_curr, q):
					v_curr = neib


	 // Query Seaarch Phase (beam search)
	add v_curr to candidates
	add v_curr to results
	mark v_curr as visited
	while candidates is not empty:
		pop element v_cand of candidates with minimum distance to q
		d_worst = maximum distance from element of results to q
		if dist(v_cand, q) > d_worst:
			break

		uneighbors = nodes adjacent to v_cand in GO that are not marked as visited
		for neib in uneighbors:
			mark neib as visited
			d_worst = maximum distance from element of results to q
			if (dist(neib, q) < d_worst) or (results has less than ef elements):
				add neib to results
				add neib to candidates
				if results has more than ef elements:
					pop element of results with maximum distance to q

	sort results in ascending order of distance from q
	return the first n_results elements of results
```

Figure 3-1: IPNSW Pseudocode.

Since GraphIt doesn't fully support all the required data structures such as priority queue for custom types and list of graphs, we have to use them as external functions in GraphIt. For example, reading input data (list of hierarchical graphs) is written in python and finding the closest points to queries is written in C++.

Since IPNSW is divided into three distinct phases, we can optimize them separately.

3.3.1 Constructing Graphs Optimizations

GraphIt internally uses GAP Benchmarking Suite [4] graph builder to construct the input graph which takes the list of Edge objects. Therefore, GraphIt provides many utility functions to convert from different graph formats to GAP format. In addition, IPNSW requires us to read input data in python CSR (Compressed Sparse Row) format [22].

In the original GraphIt version, we do the conversion from python CSR to GAP graph object sequentially as can be seen from Figure 3-2. This sequential function takes in *indptr* which is an offset array of size $|V|$ where each index indicates the starting location of edges for that index in array *indices*. Then, this function just loops through every vertex, find the start and end of edges for that vertex from *indices* array, and construct edges sequentially.

```
65  static Graph builtin_loadEdgesFromCSR(const int32_t* indptr, const NodeID* indices, int num_nodes, int num_edges) {
66          typedef EdgePair<NodeID, NodeID> Edge;
67          typedef pvector<Edge> EdgeList;
68          EdgeList el;
69          for (NodeID x = 0; x < num_nodes; x++)
70                  for(int32_t _y = indptr[x]; _y < indptr[x+1]; _y++)
71                          el.push_back(Edge(x, indices[_y]));
72          CLBase cli(0, NULL);
73          BuilderBase<NodeID> bb(cli);
74          return bb.MakeGraphFromEL(el);
75  }
```

Figure 3-2: Sequential Construction of Edges

But, the outer loop (Line 69) is easily parallelizable since each element of the loop accesses the different chunks of *indices* array and the starts and ends of those chunks are specified by *indptr* array.

To parallelize the outer loop, we need to know the exact location of the edge that we are adding into the edge list. We don't need this functionality in serial case because we could just keep inserting the edges at the end of edge list. To know the exact location, we need to maintain a prefix sum array of vertex degrees so that we know where to start looking from *indices* array. Prefix sum array can be constructed in parallel using a variant of parallel prefix sum algorithm used in GAP. Figure 3-3 shows the final code snippet. The new function does the following:

1. Construct the vertex degree array (Line 165 - 169 of Figure 3-3)

2. Does prefix sum on vertex degree array in parallel (Line 173 of Figure 3-3)

3. Create edges in parallel (Line 175 - 181 of Figure 3-3)

```
155 static Graph builtin_loadEdgesFromCSR(const int32_t* indptr, const NodeID* indices, int num_nodes, int num_edges) {
156
157     typedef EdgePair<NodeID, NodeID> Edge;
158     typedef pvector<Edge> EdgeList;
159     typedef pvector<int32_t> DegreeList;
160     EdgeList el;
161     el.resize(num_edges);
162     DegreeList dl;
163     dl.resize(num_nodes);
164
165     #pragma omp parallel for schedule(dynamic, 64)
166     for (NodeID x = 0; x < num_nodes; x++) {
167         int32_t degree = indptr[x+1] - indptr[x];
168         dl[x] = degree;
169     }
170
171     CLBase cli(0, NULL);
172     BuilderBase<NodeID> bb(cli);
173     auto prefSum = bb.ParallelPrefixSum(dl);
174
175     #pragma omp parallel for schedule(dynamic, 64)
176     for (NodeID x = 0; x < num_nodes; x++) {
177         auto startOffset = prefSum[x];
178         for(int32_t i = 0; i < dl[x]; i++) {
179             el[startOffset+i] = Edge(x, indices[startOffset + i]);
180         }
181     }
182
183     return bb.MakeGraphFromEL(el);
184 }
```

Figure 3-3: Parallel Construction of Edges

This new function is already integrated into GraphIt.

3.3.2 Greedy Walk Optimizations

In the original IPNSW code, greedy walk phase is written in Python and run sequentially for every query points. We are able to get a good performance improvement by parallelizing this phase over query points and rewriting everything in C++. For example, we are able to cut down the time from 0.61 seconds to 0.0064 seconds on 1024 queries. Since this phase is relatively straightforward, we get very good speedup on multicore machines.

3.3.3 Multiple Query optimizations

In the original IPNSW code, we find the closest points to the query by using priority queues to keep track of the shortest distances to the query. Since the initialization of those priority queues are negligible and sizes of those priority queues are constant, we can easily maintain a different priority queues for every query points. As each query has its' own priority queue, they can be easily parallelized over query points. Since GraphIt currently doesn't support custom type for priority queues, this function is implemented in C++ and called as an extern function into GraphIt.

3.4 Summary

In this chapter, we presented some of the performance optimizations we develop for Triangular Counting Algorithm, Multiple Starting Point applications, and IPNSW. For Triangular Counting, we developed different intersection methods that can be configured depending on the nature of the input. For Multiple Starting Point applications, we modified current GraphIt implementation of the vertex deduplication method for the direction optimization to make it thread safe for concurrent access. Finally, we hand-optimized the naive version of IPNSW in C++. We will present the results of above optimizations in Chapter 7.

Chapter 4

Programming Models

In this section, we introduce programming models and design decisions needed for implementing some of the graph applications. First, we talk about the design of *intersection* operator. Then we talk about the design of *functor* in GraphIt. Finally, we discuss about *parallel for* in GraphIt.

4.1 Intersection Operator

We design *intersection* operator as a part of Triangular Counting Algorithm 2.2.1 that can be scheduled through GraphIt's usual scheduling language. Table 4.1 summarizes all the intersection methods we currently support along with their advantages. You can find a detailed explanation of how those different schedules from Section 3.1.1.

Intersection Methods	When to use
Naive Intersection	When both sorted sets are small
Hiroshi Intersection	When both sorted sets are comparable in size and large
Binary Search Intersection	When sorted sets are extremely unbalanced
MultiSkip Intersection	When the intersection is sparse and one set is small

Table 4.1: Different Intersection Schedules in GraphIt

In GraphIt, *intersection* takes in four required parameters and an optional parameter ref that tells us to not count vertices labeled beyond ref. (Figure 4-1) The optional parameter is used for Triangular Counting to avoid double counting. it is

```
1   const edges : edgeset{Vertex, Vertex} = ...
2
3   func main()
4       ...
5       var src_nghs : vertexset{Vertex} = edges.getNgh(src);
6       var dst_nghs : vertexset{Vertex} = edges.getNgh(dst);
7       var src_ngh_size : uint_64 = edges.getOutDegree(src);
8       var dst_ngh_size : uint_64 = edges.getOutDegree(dst);
9
10      #s1# var value: uint_64 = intersection(src_nghs, dst_nghs, src_ngh_size,dst_ngh_size, ref=INF);
11  end
12  schedule:
13      program->configIntersection("s1", "HiroshiIntersection);
```

Figure 4-1: Intersection Operator in GraphIt

same as the internal check we discussed in 2.2.1. We have to pass in the sizes of the sets ($src\_ngh\_size, dst\_ngh\_size$) since they are internally represented as C++ arrays which don't have size information. We can also use a scoped label (#s1#) to schedule different intersections. The different intersection methods are included in GraphIt as runtime library functions that can be called directly.

4.2 Functor

In Mathematics, *functor* is defined as a map from categories to categories while *function* is a map from sets to sets. Therefore, *functor* serves as a higher order operation compared to *function*. Roughly, we can think of *functor* operation as:

1. "Unboxes" the value (special case can be a set) from the category from incoming *functor* F_{in}.

2. Maps the value to a new value using *function* f.

3. "Boxes" the result into outgoing *functor* F_{out}.

In C++, *functor* works as a function with its' own state such that it still works as a higher order *function*. *functor* is useful when we want to do certain operations only on non global data structures. For example, in the case of Betweenness Centrality, we need to maintain the count of paths that each vertex is crossed starting from one specific starting point. Therefore when running multiple starting point in parallel, it

is important to have the count of paths to be local for each starting point. To support this family of applications (application with multiple starting points), we decide to extend current GraphIt compiler to support *functor* operations. While it is certainly possible to pass the local data as extra parameters to an ordinary function, we decide it is better to implement *functor* for following reasons:

- **Parameterization**: *functors* are easy to parameterize thus making the code logic more modular.
- **Performance**: C++ *functors* can often be inlined by the compiler unlike function pointers for better performance.
- **Abstraction**: functions can be interpreted as *functors* without states (e.g. special case), therefore functor support adds another level of abstraction to the program
- **Convenience**: By default, GraphIt already generates functions as template objects with empty states. Therefore, it is easier to extend the GraphIt compiler to add state information to those template objects.

As shown in Figure 4-2, functor in GraphIt has the following syntax (highlighted in blue). All the functor states are given inside the square brackets. Notice that the body inside *functor* is working with local_score which is a local data that is initialized inside a for loop whereas the *function* is updating the globally declared variable (global_score).

4.3 Parallel For

In our initial experiments with multiple starting point applications, we discover that treating each point independently and running them in parallel gives us better performance than running them in serial. Our initial concern is that parallelizing in terms of starting points can hurt the performance because we could potentially lose some amount of parallelization within each starting point. If the inner parallelization

```
1  const global_score: vector{Vertex}(int) = 0;
2
3  func update_vertex_function(v: Vertex)
4    global_score[v] = 5;
5  end
6
7  func update_vertex_functor[local_score: vector{Vertex}(int)](v: Vertex)
8    local_score[v] = 5;
9  end
10
11 func main()
12   for i in 0:5
13     var local_score: vector{Vertex}(int) = 0;
14     vertices.apply(update_vertex_function);
15     vertices.apply(update_vertex_functor[local_score]);
16   end
17 end
```

Figure 4-2: Functor vs Function in GraphIt. Function operations are in red and Functor operations are in blue.

is too good, we won't gain much performance improvement from parallelizing the starting points. But, according to our micro benchmark, we notice that the inner parallelization can't fully take advantage of the all cores.

It is essential, however, that we need to be careful about the number of threads that are used in parallelizing over starting points. If the number of threads are too high, this will hurt the inner parallelism within each starting point because of the overhead of spawning too many threads. For example, if we are running 64 starting points in parallel, we find that using 4 threads for (e.g 16 starting points per thread) have better performance than running them using 64 threads (e.g one starting point per thread). Since the outer number of threads can potentially depend on different applications, we decide to implement a new GraphIt operator that can support running multiple starting points in parallel - *parallel for* with the option to tune the grain size (e.g outer thread numbers).

4.3.1 Design Decision

In the current state of GraphIt compiler, we have two options to implement *parallel for*.

- Simulate the *parallel for* operation with an user defined function that is applied over $VertexSubset$.

- Explicitly add *parallel for* as a GraphIt statement.

The first option suits better in GraphIt's general convention that is to define functions over edges or vertices. However, this approach makes it virtually impossible to reason about the correctness through our usual dependency analysis. Since we want to make sure that we guarantee correctness for the logic inside user defined functions, we decide that this approach is not feasible.

The second approach is against usual GraphIt convention but we can at least guarantee the correctness of user defined functions. However, we cannot guarantee correctness for the body of the *parallel for* loop. This design decision is similar to how *parfor* in MATLAB also do not execute iterations in a guaranteed order. Since we don't guarantee correctness inside *parallel for*, we implement many atomic operations such as *atomicAdd* and *compare_and_swap* to the GraphIt frontend such that users can freely use them when writing GraphIt code involving *parallel for*.

4.3.2 GraphIt Representation

Figure 4-3 shows how *parallel for* is represented in GraphIt with its' schedule (highlighted in blue). We use the label (#l1# in this case) to tune the *parallel for* performance through GraphIt scheduling language.

We currently support following scheduling option for the *parallel for*:

- **configParForGrainSize**: We find it useful to tune the grain size when running multiple starting points. For example, if we run n starting points with k grain size, there will be $\frac{n}{k}$ threads running the starting points in parallel. When nothing is supplied, the grain size will be $min(2048, ceil(\frac{n}{8 \times p}))$ where p is the number of workers created.

```
...
func main()
    #l1# par_for i in 0:64
        ...
        #s1# edges.apply(udf);
        ...
    end
end

schedule:
    program->configParForGrainSize("l1", 16)
    program->configApplyDirection("l1:s1", "SparsePush-DensePull");
```

Figure 4-3: Parallel For in GraphIt

4.4 Summary

In this chapter, we presented programming models for *intersection* operator, *functor*, and *parallel for*. We can tune *intersection* operator using GraphIt scheduling language by calling appropriate library functions based on the user schedule. In addition, we implemented *functor* which works as a function with states in GraphIt. Functors are useful when working with non global data structures. Finally, we added *parallel for* which can be tuned through grain size. (how much work each thread completes).

In the next chapter, we will discuss how we implement above features in GraphIt Compiler. We will also discuss some uses cases using these features in Chapter 6 and performance results in Chapter 7.

Chapter 5

Compiler Implementation

In this chapter, we discuss the implementation details of the newly added features in GraphIt compiler. This chapter solely focuses on how the new features implemented internally while the previous chapter focuses on the programming models to the user. For the sake of consistency, we talk about how GraphIt frontend, midend, and backend are changed for each new additions: *intersection*, *functor*, and *parallel for*.

5.1 Intersection Operator

As we discussed in 4.1, *intersection* is a GraphIt operator that can be tuned through the scheduling language. Therefore, this new addition requires changes in all major components of GraphIt Compiler.

5.1.1 Frontend

We construct a new FIR node for the *intersection* operator that stores all the meta information such as which sorted sets we intersect and sizes of those sets. We decide to keep the sizes of the sets in the argument because sets are internally represented as C++ arrays. Moreover, we register different intersection methods in the scheduling object.

5.1.2 Midend

As per GraphIt design, we construct a new MIR node for the *intersection* operator that contains exactly the same information as the corresponding FIR node along with the specified schedule. The schedule is added through the MIR Lowering pass.

5.1.3 Backend

Since our original C++ implementations of different intersection algorithms were developed on top of GAP Benchmarking Suite [4] and some part of GraphIt runtime library is inspired from GAP, we decide to use the C++ implementations as a runtime library such that the generated code can directly call them. Therefore, in the backend, we directly call the intersection method that corresponds to the user specified schedule.

Putting everything together, Figure 5-1 shows the flow of intersection operator in GraphIt.

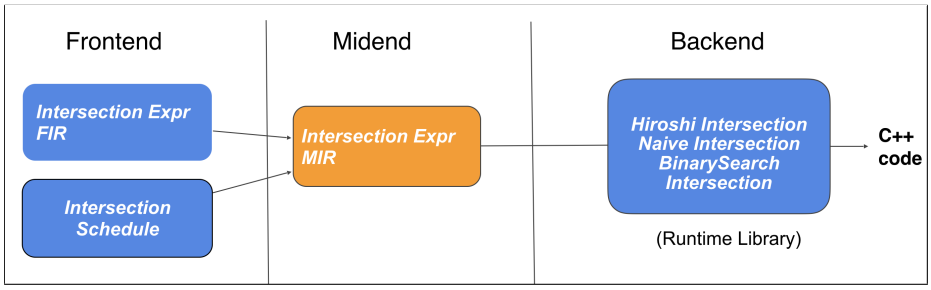

Figure 5-1: Intersection Operator Design in GraphIt

5.2 Functor

Functor is used for Multiple Starting Point applications since functor can work on starting point specific data structures unlike regular functions. In this section, we discuss how it is implemented in GraphIt compiler.

5.2.1 Frontend

Originally, GraphIt uses just the function name as the identifier for every function in the program. However, the drawback of this approach is we have no way to store the local state information of functors in the function identifier since it is represented as a plain string. Therefore, we decide to introduce a new FIR node called *FuncExpr* that include the function/functor name and all the state information with it. Since normal functions can be treated as functors without any states, *FuncExpr* work as an abstraction on top of both function and functor. (Figure 5-2)

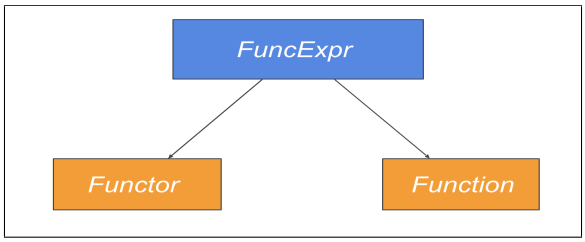

Figure 5-2: *FuncExpr* Abstraction

Moreover, GraphIt originally only used to work on globally declared arrays thus making it impossible to create local arrays inside functions. Therefore, we extend the GraphIt compiler to support initializing local arrays. Since we use *TensorArrayExpr* FIR node as an abstraction for arrays, adding local array support involves a simple changes in the parser. Specifically, we use *const* keyword for global objects and *var* keyword for local objects. Therefore, we just need to make sure that the parser function for arrays recognize *var* keyword.

5.2.2 Midend

As we discussed in the above section, we also introduce MIR node called *FuncExpr* which is pretty much identical to FIR *FuncExpr* especially since there is no schedule to attach. Therefore most of the changes involve rewriting some visitor methods to accept *FuncExpr*instead of plain strings.

GraphIt guarantees correctness inside user defined functions through its' dependency analysis in the MIR Lowering passes. This includes detecting which operations

need to be thread-safe and replacing them with atomic operations. But the original GraphIt compiler does the dependency analysis only on global arrays, therefore we extend the current analysis to support local arrays as well. In the original GraphIt dependency analysis, we find the meta information about the array operations from the context map on global arrays which cannot be extended to local arrays. Therefore, we replace this logic by relying solely on the information that is stored inside *TensorArrayExpr* MIR node which represents both local and global arrays.

5.2.3 Backend

This part involves adding new visitor methods for *FuncExpr* in the backend and rewriting some methods to accept *FuncExpr* instead of plain function identifier strings.

Putting everything together, Figure 5-3 summarizes how *functor* is generated in GraphIt. First, both *function* and *functor* are parsed into a FIR node *FuncExpr*. Then, we create a MIR node *FuncExpr* and pass it through dependency analysis lowering pass to add atomic operators if necessary. Finally, we generate a C++ functor.

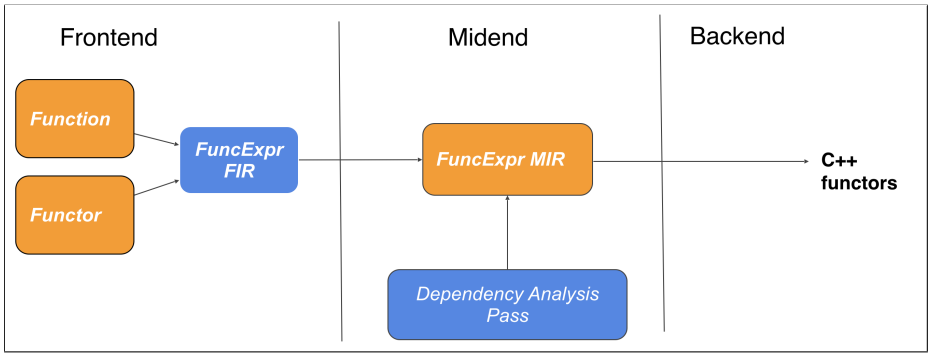

Figure 5-3: Functor Design in GraphIt

In Figure 5-4, we see that an array *a* is initialized as a field to a functor *addOne* and the main operation of the functor *addOne* operates only on the field array *a*.

```
1  func addOne[a: vector{Vertex}(int)](v: Vertex)
2
3      a[v] += 1;
4
5  end
6
```

```
1  struct addOne {
2    int  * a;
3    addOne(int  * a): a(a) {}
4    void operator() (NodeID v) {
5      a[v] += (1) ;
6    };
7  };
```

Figure 5-4: Functor in GraphIt vs Functor in C++. Left is the GraphIt code and right is the generated C++ code.

5.3 Parallel For

We implement *parallel for* as a way to parallelize over starting points in GraphIt. In GraphIt, users can configure *parallel for*'s grain size for better performance. In this part, we discuss how we integrate it to GraphIt compiler.

5.3.1 Frontend

Similar to other additions, we create a new FIR node called *ParForStmt* which contains the body of the *parallel for*, the loop domain, the loop variable, and the statement label. We also register a new scheduling object which specifies the grain size for the parallel for loop discussed in 4.3.2. The scheduling object is similar to OPENMP/CILK in that we tune the grain size to achieve better performance.

5.3.2 Midend

Similar to the Frontend changes, we create a new MIR node called *ParForStmt* which contains all the meta information along with the scheduling parameters that is added through the MIR Lowering Pass.

5.3.3 Backend

As *parallel for* is somewhat an independent feature, the backend changes only involve adding a visitor function for *ParForStmt*. We use *cilk_for* macro from CILK for tuning the *parallel for*.

In Figure 5-5, we show how a simple GraphIt *par_for* snippet translates into C++ code. We use grain size of 4 in the schedule which directly translates to CILK grain

```
1  func main()
2      #l1# par_for i in 0:5
3          simpleArray[i] = 5;
4      end
5  end
6  schedule:
7      program->configParForGrainSize("l1", 4);
```

```
1  #pragma cilk grainsize = 4
2  cilk_for( int i = (0) ; i < (5) ; i++ )
3  {
4    simpleArray[i] = (5) ;
5  }
6
```

Figure 5-5: Parallel For in GraphIt vs Functor in C++. Left is the GraphIt code and right is the generated C++ code.

size of 4.

5.4 Summary

In this chapter, we discuss how *intersection* operator, *functor*, and *parallel for* are implemented in GraphIt compiler. For *intersection* operator, we introduce *IntersectionExpr* FIR, MIR nodes to represent it internally. For *functor*, we introduce *FuncExpr* abstraction for both *functor* and *function* and rewrite significant portion of GraphIt compiler visitor functions to accept *FuncExpr*. Finally, we add *ParForStmt* to represent *parallel for* in GraphIt.

In the next chapter, we show how all the new features work together to implement certain applications.

Chapter 6

Use Cases

In this chapter, we present two applications - Triangular Counting and Closeness Centrality as use cases for our new features. Triangular Counting uses *intersection* operator while Closeness Centrality uses both *functor* and *parallel for*.

6.1 Triangular Counting in GraphIt

Putting everything together, Triangular Counting algorithm can be implemented in GraphIt shown in 6-1.

In Line 24, we relabel the vertices by degree. This is useful when the average degree is high enough and and if the degree distribution is dense power-law graph. By doing so, we put neighboring vertices adjacent to each other in term of their vertex IDs such that they can be be accessed in a cache efficient manner. This method is heavily inspired by GAP [4].

In Line 25, we apply *incrementing_ count* method, a bulk of our Triangular Counting Algorithm, over the edges of the graph which is then parallelized over source vertices through the GraphIt scheduling language. This can be seen from the label #$s1$# and Line 32.

In Line 7, we intersect the neighboring vertices of both *src* and *dst* for each edge. We implement a check $dst > src$ to ensure we count each triangle only once. Without this check, each triangle would be double counted since Triangular Counting works

```
element Vertex end
element Edge end
const edges: edgeset{Edge}(Vertex,Vertex) = load(argv[1]);
const vertices: vertexset{Vertex} = edges.getVertices();
const triangles: uint_64 = 0;

func incrementing_count(src: Vertex, dst: Vertex)
        if dst < src
                var dst_ngs : vertexset{Vertex} = edges.getNgh(dst);
                var src_ngs : vertexset{Vertex} = edges.getNgh(src);
                var dst_ngh_size: uint_64 = edges.getOutDegree(dst);
                var src_ngh_size: uint_64 = edge.getOutDegree(src);
                #s2# var value: uint_64 = intersection(src_nghs, dst_nghs, src_ngh_size, dst_ngh_size);
                vertexArray[src] = value;
        end

end

func reset(v: Vertex)
        vertexArray[v] = 0;
end

func main()
        edges = edges.relabel();
        #s1# edges.apply(incrementing_count);
        triangles = vertexArray.sum();
        vertices.apply(reset);
        print triangles;
end

schedule:
        program->configApplyDirection("s1", "SparsePush")
               ->configApplyParallelization("s1", "dynamic-vertex-parallel", 64);
        program->configIntersection("s2", "HiroshiIntersection");
```

Figure 6-1: Triangular Counting in GraphIt

on undirected graphs. #*s*2# is used to tune the *intersection* operator. We also store the number of triangles involving *src* in *vertexArray* and sum all the elements of *vertexArray* in parallel. By doing so, we eliminate synchronization overhead because each vertex of the parallel region touches different part of *vertexArray*. Summing *vertexArray* in parallel takes a negligible amount of time compared to the Triangular Counting Algorithm.

6.2 Closeness Centrality in GraphIt

We discuss the Closeness Centrality Algorithm with multiple starting points on unweighted graphs. A brief description of how this algorithm works can be found in 2.2.3. In our version of Closeness Centrality for unweighted graphs, there are multiple starting points where each starting point runs a direction-optimized BFS [3] to keep track of distances of each vertex from itself. To further speed up our program, we run each starting point in parallel such that each starting point maintain it's local data to keep track of distances to other vertices. By making each start point to work on their own local data, we reduce the synchronization overhead.

In Figure 6-2, we initialize a local array *checked_local* for each starting point in Line 22 and all the operations which are related to BFS work on *checked_local* (Lines 23-36). Then, we just sum values in *checked_local* to obtain the score for the specific starting point (Lines 38 - 41). Note that we are heavily using GraphIt functors over a local array *checked_local* throughout the program. We also free *checked_local* array as soon as we are done working on it (Line 41) to make sure other threads which are running different starting points can pick up the recently freed space for better performance.

To run the starting points in parallel, we make use of *parallel for* defined as *par_for* in GraphIt. The usage of *parallel for* can be seen in Line 20 of Figure 6-2. The label #**l1**# indicates that we can tune the *parallel for* for this specific use case.

Figure 6-3 shows the schedule that is used for the Closeness Centrality Algorithm. The first schedule (Line 50) is used to tune the *parallel for* thus each thread pro-

cesses 16 starting points. The second schedule (Line 52-54) is used for the direction-optimized BFS within each starting point. Note that *l1:s1* label ensures that the direction-optimized BFS is within the *parallel for* scope.

```
1  element Vertex end
2  element Edge end
3
4  const edges : edgeset{Edge}(Vertex, Vertex) = load (argv[1]);
5  const vertices : vertexset{Vertex} = edges.getVertices();
6
7  const scores: vector{Vertex}(int) = 0;
8
9  func updateEdge[checked_local: vector{Vertex}(int)](src : Vertex, dst : Vertex)
10   checked_local[dst] = checked_local[src] + 1;
11 end
12
13 func toFilter[checked_local: vector{Vertex}(int)](v : Vertex) -> output : bool
14   output = checked_local[v] == -1;
15 end
16
17 func main()
18
19   var starting_points: vector[64](int) = {1, 2, …,64};
20   #l1# par_for i in 0: 64
21     var checked_local : vector{Vertex}(int) = -1;
22     var start_vertex : int = starting_points[i];
23     checked_local[start_vertex]  = 0;
24     var frontier : vertexset{Vertex} = new vertexset{Vertex}(0);
25     frontier.addVertex(start_vertex);
26     while (frontier.getVertexSetSize() != 0)
27
28       #s1# var output : vertexset{Vertex} = edges.from(frontier)
29                                               .to(toFilter[checked_local])
30                                               .applyModified(updateEdge[checked_local], checked_local);
31
32       delete frontier;
33       frontier = output;
34
35     end
36     delete frontier;
37
38     var notConnected : vertexset{Vertex} = vertices.filter(toFilter[checked_local]);
39     var amountNotConnected : int = notConnected.getVertexSetSize();
40     var sum: int = checked_local.sum();
41     sum = sum + amountNotConnected;
42
43     scores[start_vertex] = sum;
44
45     delete checked_local;
46
47 end
```

Figure 6-2: Unweighted version of Closeness Centrality

```
49 schedule:
50     program->configParForGrainSize("l1", 16);
51
52     program->configApplyDirection("l1:s1", "SparsePush-DensePull")
53            ->configApplyParallelization("l1:s1", "dynamic-vertex-parallel")
54            ->configApplyDenseVertexSet("l1:s1","bitvector", "src-vertexset", "DensePull");
```

Figure 6-3: Schedules for Closeness Centrality UnWeighted

6.3 Summary

In this chapter, we present two applications as use cases for our new additions to GraphIt. Triangular Counting algorithm uses our new operator - *intersection* which can be tuned through GraphIt scheduling language. Closeness Centrality algorithm uses *functor* and *parallel for*. We use *functor* to operate on start point specific data structures and *parallel for* for parallelizing over starting points.

In the next chapter, we present the whole evaluation of our new additions to GraphIt based on graph algorithms discussed in 2.2.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 7

Evaluation

7.1 Machine Description

For the evaluation of our work, we use a dual-socket system with Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores and 48 hyper-threads. Each core has the clock speed of 1.20 GHz. The system has 128 GB of DDR3-1600 memory and 30 MB last level cache on each socket and runs with Transparent Huge Pages (THP) enabled and Ubuntu 18.04.

7.2 Dataset Description

The evaluation of Intersection Operator (Triangular Counting) uses five graphs: Road-USA, Twitter, Web, Kron, and Urand graphs. This is in accordance with GAP Benchmark Suite [4].

The evaluation of Multiple Starting Point applications uses ten graphs. Note that we don't use all ten of them in some applications such as Closeness Centrality Weighted since the weighted version of those graphs are not available.

Table 7.1 summarizes the descriptions of the graphs that are used in both Triangular Counting and Multiple Starting Point applications.

The evaluation of IPNSW uses a graph provided from annual DARPA TA2 evaluation. The graph contains one million vertices and 16 million edges.

Name	Description	Vertices (M)	Edges (M)	Degree	Degree Dist	Diameter	References
Road-USA	Roads of USA	23.9	57.7	2.4	bounded	6,304	[8]
Twitter	Twitter Follow Links	61.6	1,468.4	23.8	power	14	[14]
Web	Web Crawl of .sk Domain	50.6	1,930.3	38.1	power	135	[5]
Kron	Kronecker Synthetic Graph	134.2	2,111.6	15.7	power	6	[20, 16]
Urand	Uniform Random Graph	134.2	2,147.5	16.0	normal	7	[10]
socLive	LiveJournal Community	4.85	68.99	16.0	power	16	[21]
com_orkut	Orkut Community	3.07	117.2	16.0	power	9	[25]
Road-Germany	Roads of Germany	11.5	12.4	2	bounded	0	[21]
Road-CA	Roads of CA	1.96	2.77	16.0	bounded	849	[15]
Road-TX	Roads of TX	1.38	1.92	16.0	bounded	1054	[15]
Rmat17	Recursive Matrix Model	0.13	3.73	28.4	power	8	[7]

Table 7.1: Graphs used for evaluation

7.3 Results

In this section, we measure performance as a total execution time of the program. We compile all of our programs with GNU C++ Compiler version 7.5.0. First, we present results of the new *intersection* operator in Triangular Counting algorithm. Then we discuss the results of multiple starting point applications which are implemented using *functor* and *parallel for*. Finally, we reveal the optimization results for IPNSW.

7.3.1 Intersection Operator

In this part, we present the best performance numbers of Triangular Counting Algorithm in GraphIt versus the GAP Benchmarking Suite [4]. The results are summarized in Table 7.2. We find that *HiroshiIntersection* is the fastest intersection method on Kron, Twitter, and Urand graphs since they have many vertices that have huge neighboring list of vertices. On the other hand, we use *NaiveIntersection* for the Road-USA and Web graphs since the average degree of those graphs are low making many of the vertices to have a small number of vertices. Even though we use same intersection method as GAP on Road-USA and Web graphs, we see some slowdown due to framework overhead.

Moreover, we notice that *MultiskipIntersection* method performs almost as good as *HiroshiIntersection* on Urand graph because the number of intersections is very sparse. (Urand only has 5,378 triangles) However, *MultiskipIntersection* is significantly slower on other graphs as they have more triangles compared to Urand graph.

We also see that *BinarySearchIntersection* is much slower than other intersection methods because it works the best when the set sizea are extremely unbalanced which occurs very rarely.

Graphs	GAPBS	Hiroshi	BinarySearch	Multiskip	Naive	# of triangles
Web	20.35	20.62	239.20	26.77	**19.22**	84,907,041,475
Road-USA	0.036	0.057	0.067	0.047	**0.037**	438,804
Kron	300.63	**281.03**	—	405.82	299.16	106,873,365,648
Twitter	68.17	**63.678**	—	76.70	69.96	34,824,916,864
Urand	18.22	**17.46**	47.78	17.90	18.84	5,378

Table 7.2: Triangular Counting Benchmarking Results. Highlighted ones are the fastest numbers and each column name is the name of the intersection method we use. Kron and Twitter numbers for *BinarySearch* intersection are omitted since they were too slow. All numbers except the last column (number of triangles in a graph) are recorded in seconds.

7.3.2 Multiple Starting Points

In this part, we present how much improvement we gain from running multiple starting points in parallel. For the baseline, we use the version that each starting points are run sequentially. In other words, the baseline numbers only parallelize within each starting point. For the consistency purposes, every program is compiled with CILK and all the numbers are reported in seconds. For each graph application, we use 64 starting points that are chosen randomly. In addition, we refer to parallelization within each starting point as *inner parallelism* and parallelization over starting points as *outer parallelism*. After presenting the results for each application, we conclude with some common takeaways.

Betweenness Centrality

Table 7.3 summarizes the result. As we can see from the table, we are almost 3.4x - 15x faster on road graphs (Road-USA, Road-Germany, Road-CA, Road-TX) and 2.6x faster on Rmat17 graph. This result agrees with our initial benchmark discussed in 2.4.1 that road graphs have a bad inner parallelism. For Rmat17 graph, we also get good speedup because it is a relatively small graph thus not benefiting from inner parallelism as the frontier sizes in the forward run of Betweenness Centrality are always small.

On the other hand, we get some mild improvement on socLive and Web graphs (around 13%) and no improvement on Kron and Twitter graphs. We don't get improvements on Kron and Twitter graphs because they already have high inner parallelism. However, socLive and Web graphs have worse inner parallelism than Kron and Twitter therefore we are able to get some improvements by adding outer parallelism.

Another observation we can make is that we use more outer threads on road graphs and small graphs like Rmat17 (8 - 32 threads) than social and big graphs (2 - 4 threads). As we can see from Figure 7-2, we get even more speedup as we increase the number of threads on road graphs. For relatively bigger graphs such as Road-USA and Road-Germany, the performance saturates around 8 - 16 outer threads while we get the best performance for smaller road graphs even after 16 threads. These observations support our point that road graphs and small graphs don't benefit from inner parallelism such that using few threads is as effective as using many threads within each starting point.

In the case of social graphs, we can see Kron and Twitter graphs significantly slow down as we increase the number outer threads (Figure 7-1). On the other hand, Web and socLive graphs tend to speed up the most around 2 - 4 outer threads and slow down afterwards. In addition, we can see that Web graph performance doesn't change as much as others as we increase the number of outer threads.

Graphs	Inner Parallelism, s	Inner & Outer Parallelism, s	Outer Threads	Speedup
Web	69.90	61.87	4	1.13
Road-USA	217.60	30.92	32	7.04
Kron	240.95	266.94	2	0.90
Twitter	92.15	91.01	2	1.01
socLive	4.49	3.98	4	1.13
Rmat17	0.31	0.12	16	2.66
Road-Germany	82.66	24.48	8	3.38
Road-CA	18.21	1.60	32	11.40
Road-TX	19.51	1.29	32	15.12

Table 7.3: Betweenness Centrality Benchmark Results.

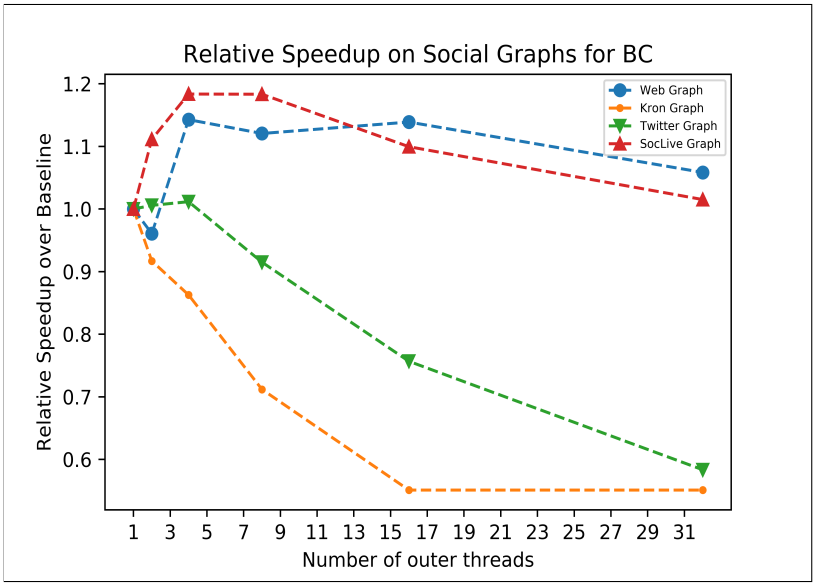

Figure 7-1: Relative Speedup vs Number of Outer Threads for BC on social graphs

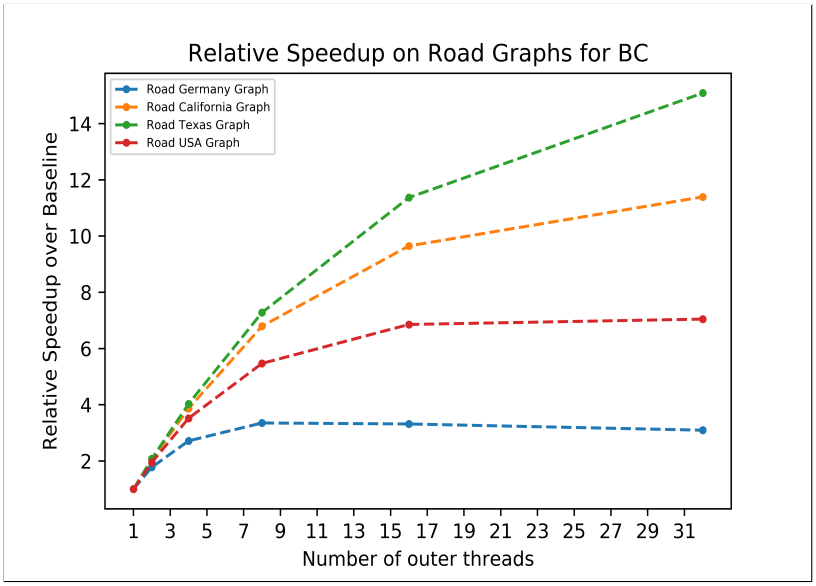

Figure 7-2: Relative Speedup vs Number of Outer Threads for BC on road graphs

Closeness Centrality Unweighted

Table 7.4 summarizes the benchmark result of the unweighted version of Closeness Centrality. In general, we observe similar characteristics as Betweenness Centrality benchmark results. We have a very good speedup on road Graphs (Road-USA, Road-Germany, Road-CA, and Road-TX) and small graphs such as Rmat17. On the other hand, we don't get any improvement on Kron and Web graphs because they already have good inner parallelism. In addition, we get some improvements on socLive and Twitter graphs.

Furthermore, we get the best performance using around 16 - 32 threads on outer parallelism on road graphs and 8 threads for small graphs such as socLive and Rmat17 graphs (Figure 7-4) Yet we use 2 - 4 threads on the rest of the graphs. In Figure 7-3, all social graphs except Kron graph speeds up the most around 2 - 4 threads and slow down afterwards. We can also see that Web graph performance stays relatively the same as we increase the number of outer threads.

Graphs	Inner Parallelism, s	Inner & Outer Parallelism, s	Outer Threads	Speedup
Web	35.34	34.41	4	1.03
Road-USA	117.71	14.86	32	7.92
Kron	36.92	36.83	2	1.00
Twitter	23.74	20.08	4	1.18
socLive	1.90	1.37	8	1.39
Rmat17	0.19	0.07	8	2.66
Road-Germany	43.51	10.07	16	4.32
Road-CA	10.10	0.85	32	11.85
Road-TX	10.67	0.73	32	14.68

Table 7.4: Closeness Centrality Unweighted Benchmark Results.

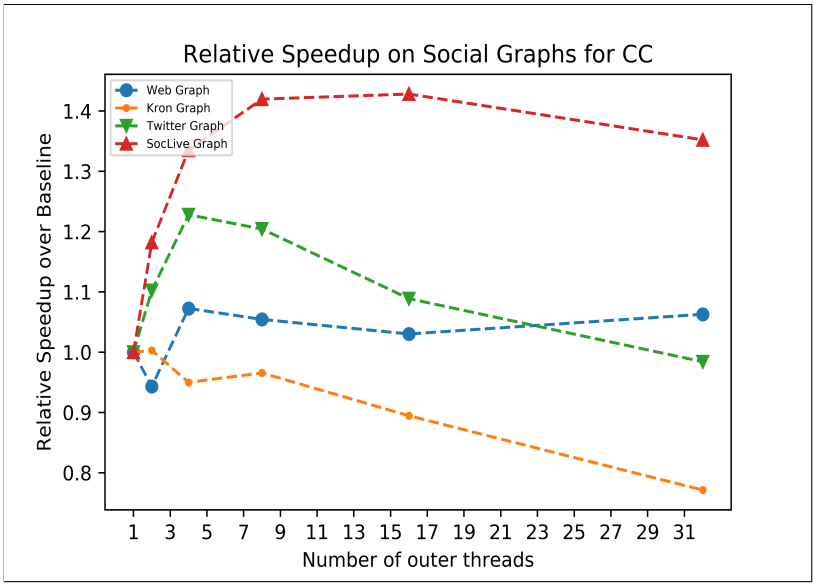

Figure 7-3: Relative Speedup vs Number of Outer Threads for Closeness Centrality on social graphs

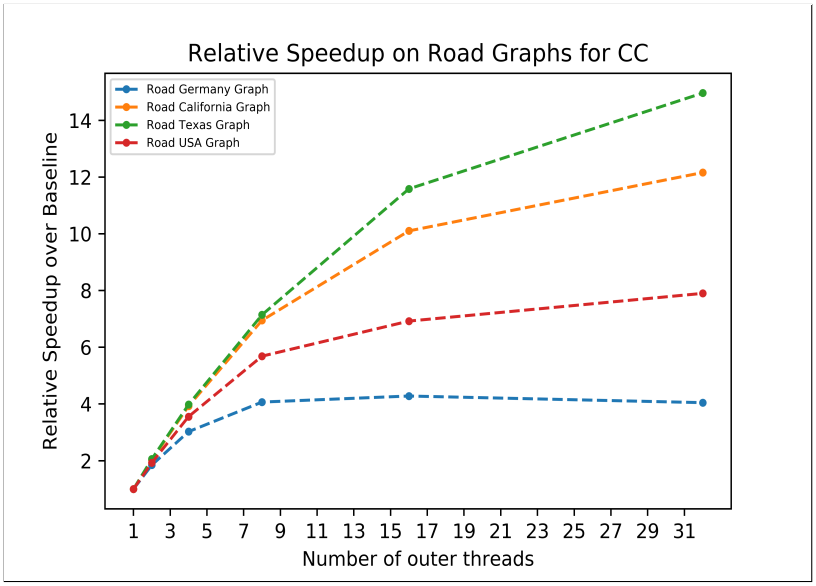

Figure 7-4: Relative Speedup vs Number of Outer Threads for Closeness Centrality on road graphs

Closeness Centrality Weighted

Table 7.5 summarizes the result for the weighted version of Closeness Centrality. Note that we only use five graphs in this part since some graphs don't have the weighted versions. Since we don't have the weighted versions of road graphs, we use Rmat17 as an example of a graph which has bad inner parallelism and we can see that there is almost 2x speedup when we add outer parallelism. For social graphs, we have some mild improvements. As usual, we don't get a speedup on Kron graph due to its' already high inner parallelism.

Graphs	Inner Parallelism, s	Inner & Outer Parallelism, s	Outer Threads	Speedup
web	140.21	109.30	8	1.28
kron	249.23	263.60	2	0.95
twitter	93.06	92.10	2	1.01
socLive	4.44	4.05	8	1.10
Rmat17	0.27	0.13	8	2.10

Table 7.5: Closeness Centrality Weighted Benchmark Results.

PR-Nibble

Table 7.6 summarizes the result for PageRank-Nibble. Unlike the other applications, we don't get very good speedup. This is because PR-Nibble is inherently hard to parallelize. PR-Nibble runs for many rounds and it maintains vertex frontiers that reduce in size after every round. Therefore, PR-Nibble achieves a very good parallelization in the beginning when there are many vertices in the frontier and it suffers from parallelization afterwards when there are very few vertices in the frontier. Since this transition happens very quickly, the overall program doesn't get good parallelism. As a result, outer parallelism doesn't help much.

Another interesting observation is that we have to use more outer CILK threads to achieve better performance than other applications on all graphs. For example, we normally find using two threads for Twitter graph is the best in other application but we use 16 threads in PR-Nibble. This is related to the point we make earlier that PR-Nibble doesn't achieve good parallelization.

Graphs	Inner Parallelism, s	Inner & Outer Parallelism, s	Outer Threads	Speedup
Web	32.73	27.27	32	1.20
Road-USA	24.44	19.79	16	1.23
Kron	33.06	34.53	16	0.96
Twitter	26.96	21.38	16	1.26
socLive	2.53	2.23	16	1.13
Rmat17	1.90	0.94	32	2.04
Road-Germany	10.17	9.52	16	1.07
Road-TX	1.90	0.94	4	2.04

Table 7.6: PageRank-Nibble Benchmark Results.

Common Takeaways

In this part, we summarize our results from previous sections and comment on observations we made.

We notice that Road graphs and small graphs benefit the most from outer parallelism since they don't scale well from inner parallelism. In some extreme cases,

we are able to get 16x speedup over the baseline. Road graphs are known to have high diameters and low average degrees so many applications tend to run for many rounds to converge and each round works with small number of vertices making it hard parallelize.

In addition, Kron graph does not benefit from outer parallelism. This is because Kron graph already achieves high inner parallelism for BC, unweighted CC, and weighted CC. For PR-Nibble, the overall program doesn't benefit too much from parallelization in general.

Furthermore, we notice that the most optimal configuration for outer threads in large graphs (social graphs in our case) is $2 \sim 4$ threads while road graphs work well with $16 \sim 32$ threads.

While tuning the number of outer threads for Web graph on all applications, we notice that Web graph doesn't get significantly slower or faster with respect to outer threads. This can be explained by the fact that Web graph has good **locality** and **similarity** between vertices [5]. Good locality implies that vertices are clustered around a common prefix making it easy for threads to work on same number of vertices. Good similarity implies that threads can roughly do equal work leading to a better load balance.

7.3.3 IPNSW

In this section, we talk about the performance improvements we get on IPNSW which is summarized in Table 7.7. We use 1024 query points and we return 10 closest points to each query as per DAPRA SDH TA2 evaluation [1].

As we mentioned in Section 3, the IPNSW algorithm is divided into three main steps and each steps can be optimized separately.

In the first phase, we deal with constructing graphs to be used by GraphIt from python CSR format. We are able to get almost 5x speedup by parallelizing the edge construction. The more in detail discussion about this optimization can be found at 3.3.1.

In the second phase, we greedily traverse the graphs to find the appropriate graph

to search for the closest points for the query. As per our discussion in 3.3.2, we rewrite the original implementation in C++ and parallelize over query points. Since this phase is straightforward, we get a very good speedup. The original python implementation takes about 0.61 seconds while the our optimized version takes 0.0064 seconds.

In the last phase, we find the 10 closest points to each queries. The original implementation is implemented serially, but it is easy to parallelize and the overhead of maintaining additional query specific local data structures are negligible compared to the main computation. By parallelizing over query points, we are able to cut down the computation time from 0.42 seconds to 0.058 seconds. For the parallelization, we just use OPENMP with a grain size of 64. We don't need to carefully tune the grain size for this application because finding the closest points to each query is roughly the same and there is no inner parallelization.

To sum up, we are able to cut down the original implementation runtime from 2.32 seconds to 0.349 seconds. (roughly 7x speedup) Note that since GraphIt doesn't support custom type priority queues, majority of IPNSW are implemented as extern C++ functions in GraphIt.

	Original, seconds	Optimized, seconds	Speedup
Graph Construction Phase	1.29	0.28	4.6
Hierarchical Greedy Walk Phase	0.61	0.0064	95
Query Phase	0.42	0.058	7.2
Total	2.32	0.35	6.6

Table 7.7: IPNSW results on 1024 concurrent queries.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 8

Conclusion & Future Work

8.1 Conclusion

In this thesis, we added several new features to GraphIt that allowed us to extend GraphIt's current use cases. Specifically, we added *intersection* operator which is widely used for Triangular Counting Algorithm. Our version of Triangular Counting Algorithm is faster than the reference implementation given by GAP Benchmarking Suite. We also added *functor* and *par_for* to support multiple starting point applications which get up to 16x speed up over the GraphIt implementations without the new features. Finally, we did some performance optimzations for IPNSW that achieved around 7x speedup over the naive implementation on the evaluation dataset.

8.2 Future Work

There are number of future directions that this thesis can take. Firstly, we can further extend GraphIt compiler to support custom types and more general array operations (for example, array of graphs) which will allow us to implement IPNSW completely in GraphIt. By doing so, we might be able to achieve even better performance. Secondly, we can further improve GraphIt's dependency analysis to detect potential atomic operations from nested user defined functions. Lastly, we can integrate our newly added features into the ongoing development of GraphIt GPU support.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Software defined hardware (sdh). https://www.darpa.mil/program/softwaredefined-hardware.

[2] R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 475–486, 2006.

[3] S. Beamer, K. Asanovic, and D. Patterson. Direction-optimizing breadth-first search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2012.

[4] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[5] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. *WWW*, pages 595–601, 2004.

[6] Ulrik Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25, 03 2004.

[7] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. volume 6, 04 2004.

[8] 9th DIMACS implementation challenge - Shortest paths. `http://www.dis.uniroma1.it/challenge9/`, 2006.

[9] Alessandro Epasto, Silvio Lattanzi, Vahab S. Mirrokni, Ismail Sebe, Ahmed Taei, and Sunita Verma. Ego-net community mining applied to friend suggestion. In *Proceedings of VLDB*, 2016.

[10] Paul Erdős and Alfréd Rényi. On random graphs. I. *Publicationes Mathematicae*, 6:290–297, 1959.

[11] Kimon Fountoulakis, David F Gleich, and Michael W Mahoney. A short introduction to local graph clustering methods and software. *arXiv preprint arXiv:1810.07324*, 2018.

[12] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. Faster set intersection with simd instructions by reducing branch mispredictions. *Proc. VLDB Endow.*, 8(3):293–304, November 2014.

[13] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. Simit: A language for physical simulation. *ACM Trans. Graph.*, 35(2):20:1–20:21, May 2016.

[14] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? *WWW*, 2010.

[15] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[16] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. *PKDD*, 2005.

[17] Haibin Liu, Karin M. Verspoor, Donald C. Comeau, Andrew MacKinlay, and W. John Wilbur. Generalizing an approximate subgraph matching-based system to extract events in molecular biology and cancer genetics. In *BioNLP at ACL*, 2013.

[18] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020.

[19] Stanislav Morozov and Artem Babenko. Non-metric similarity graphs for maximum inner product search. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 4726–4735, Red Hook, NY, USA, 2018. Curran Associates Inc.

[20] Richard C. Murphy, Kyle B. Wheeler, Brian W Barrett, and James A. Ang. Introducing the Graph 500. In *Cray User's Group*. CUG, 2010.

[21] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[22] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[23] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.

[24] Lei Wang, Fan Yang, Liangji Zhuang, Huimin Cui, Fang Lu, and Xiaobing Feng. Articulation points guided redundancy elimination for betweenness centrality. *ACM SIGPLAN Notices*, 51:1–13, 02 2016.

[25] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[26] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. Optimizing ordered graph algorithms with graphit. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 158–170, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.