Understanding and Improving Graph Algorithm Performance

by

Scott Beamer III

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Krste Asanović, Co-chair
Professor David Patterson, Co-chair
Professor James Demmel
Professor Dorit Hochbaum

Fall 2016

Understanding and Improving Graph Algorithm Performance

Abstract

Understanding and Improving Graph Algorithm Performance

by

Scott Beamer III

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Krste Asanović, Co-chair

Professor David Patterson, Co-chair

Graph processing is experiencing a surge of renewed interest as applications in social networks and their analysis have grown in importance. Additionally, graph algorithms have found new applications in speech recognition and the sciences. In order to deliver the full potential of these emerging applications, graph processing must become substantially more efficient, as graph processing's communication-intensive nature often results in low arithmetic intensity that underutilizes available hardware platforms.

To improve graph algorithm performance, this dissertation characterizes graph processing workloads on shared memory multiprocessors in order to understand graph algorithm performance. By querying performance counters to measure utilizations on real hardware, we find that contrary to prevailing wisdom, caches provide great benefit for graph processing and the systems are rarely memory bandwidth bound. Leveraging the insights of our workload characterization, we introduce the Graph Algorithm Iron Law (GAIL), a simple performance model that allows for reasoning about tradeoffs across layers by considering algorithmic efficiency, cache locality, and memory bandwidth utilization. We also provide the Graph Algorithm Platform (GAP) Benchmark Suite to help the community improve graph processing evaluations through standardization.

In addition to understanding graph algorithm performance, we make contributions to improve graph algorithm performance. We present our direction-optimizing breadth-first search algorithm that is advantageous for low-diameter graphs, which are becoming increasingly relevant as social network analysis becomes more prevalent. Finally, we introduce propagation blocking, a technique to reduce memory communication on cache-based systems by blocking graph computations in order to improve spatial locality.

To my family

Contents

List of Figures

List of Tables

Acknowledgments

Many people enriched the amazing learning experience I had during my years at UC Berkeley.

First, I would like to thank my co-advisors Krste Asanović and David Patterson. They have been a great advising team, and together, they provided me with wonderful opportunities and invaluable lessons. Krste gave me tremendous freedom and encouraged me to go out and measure and build things. Dave repeatedly taught me not only the lesson of keeping things simple, but also the importance of being able to simply explain an idea. Difficulty explaining an idea simply is an indication that more effort may be required by the explainer to distill the key insight. I would also like to acknowledge the gracious assistance from the rest of my dissertation committee members, James Demmel and Dorit Hochbaum. Jim provided useful information on sparse linear algebra, and Dorit provided helpful critiques.

I'm especially indebted to the great colleagues I had the pleasure of working with throughout my studies. Sarah Bird and Henry Cook were the other computer architecture students in my incoming cohort, and I could not have asked for better company during the long journey. The other Cosmic Cube co-inhabitants Andrew Waterman and Chris Celio provided camaraderie, thoughtful advice, and essential feedback on early ideas. The architecture group has a contagious energy, and I want to acknowledge the other barchitects: Rimas Avižienis, Yunsup Lee, Eric Love, Martin Maas, Adam Izraelevitz, Colin Schmidt, Palmer, Dabbelt, Sagar Karandikar, Donggyu Kim, Jack Koenig, David Biancolin, Albert Magyar, Brian Zimmer, and Ben Keller. The senior students Zhangxi Tan and Heidi Pan set great examples and offered useful advice.

The ParLab introduced me to superb colleagues. Chris Batten was a fantastic mentor and collaborator during the beginning of my graduate studies. Andrew Gearhart provided crucial expertise on the inner workings of Intel performance counters. Shoaib Kamil was a terrific collaborator on the SEJITS effort. Sam Williams provided a deep understanding of multicore performance optimizations. My interactions with Leo Meyerovich led to my dissertation topic. I originally wanted to build a web browser hardware accelerator, but the tree traversals within Leo's parallel page layout code motivated me to consider graph algorithms.

I am grateful to have been a member of both the ParLab and the ASPIRE Lab and for the financial support their sponsors provided. The labs have a sensational support staff that eased or even enabled much of my research. Kostadin Ilov and Jon Kuroda's technical support was indispensable, and they gracefully handled odd requests and pre-production hardware with ease. Tami Chouteau and Roxana Infante skillfully kept the labs and all other matters running smoothly. I am also appreciative of the advice and collaborations provided by other faculty within the labs from Armando Fox, Jonathan Bachrach, and Vladimir Stojanović.

I am thankful for the patience and willingness to help from the members of the graph research community. Aydın Buluç has been a great collaborator and resource for learning more about graph algorithms and their application. Kamesh Madduri taught me much about

graph processing, especially when I was just getting started. Jason Riedy continually teaches me new things about graphs and provided access to the mirasol platform.

I want to recognize those that helped guide me throughout my time at Berkeley, including the time before I even attended graduate school. Dan Garcia infected me with not only an excitement for teaching, but also an excitement for computer architecture, which became the focus of my graduate studies. Sheila Humphreys has always been a wonderful resource and a great supporter.

I am also grateful for support from the community outside of the university. Friends and ultimate frisbee teammates helped make everything more worthwhile. I want to thank Zeph Landau for being a great teammate, coach, friend, and mentor. My family has always placed a high value on education, and they provided me with continual encouragement, especially when it was most needed. Kim Long provided both resiliency and inspiration, and getting to know her has been the best part of the last three years.

Chapter 1

Introduction

Graphs represent connections, and graphs can be used to represent many types of connections in the real world, whether it be friendships between humans or road segments between street intersections. The strength of the graph abstraction is that it allows the same graph algorithms to be reused to process a diverse range of graphs, thus enabling a diverse range of applications. Although the graph abstraction has been used for centuries, there is a renewed interest in graph processing driven in large part by emerging applications in social network analysis [91, 110, 170], science [132], and speech and image recognition [96, 174]. To better support these increasingly relevant graph processing applications, research is ongoing at all levels, including applications, algorithms, implementations, frameworks, and even hardware platforms.

Unfortunately, graph algorithms are notoriously difficult to execute efficiently on current processors [98], and so there has been considerable recent effort to improve the performance of processing large graphs. The connection-centric nature of graph processing results in a communication-centric workload that often has low arithmetic intensity. On a shared memory multiprocessor, graph processing can often simultaneously underutilize both the compute throughput and the memory bandwidth. Such low utilization provides a tremendous opportunity, as it increases the potential speedup possible with a better-utilized similarly-sized hardware platform.

In this dissertation, we strive to understand graph algorithm performance in order to improve performance. To understand graph algorithm performance, we characterize graph processing workloads on real hardware. By understanding how graph algorithm software implementations interact with hardware platforms, we can improve efficiency. Changes to an algorithm can not only impact the amount of algorithmic work, but also change which hardware bottlenecks constrict performance. To help others improve graph algorithm performance via a better understanding of performance, we leverage our workload characterization to contribute a benchmark suite and a simple performance model to reason across abstraction layers. To improve performance, we also provide a novel breadth-first search algorithm that is advantageous for low-diameter graphs and an implementation technique to reduce memory communication on cache-based systems.

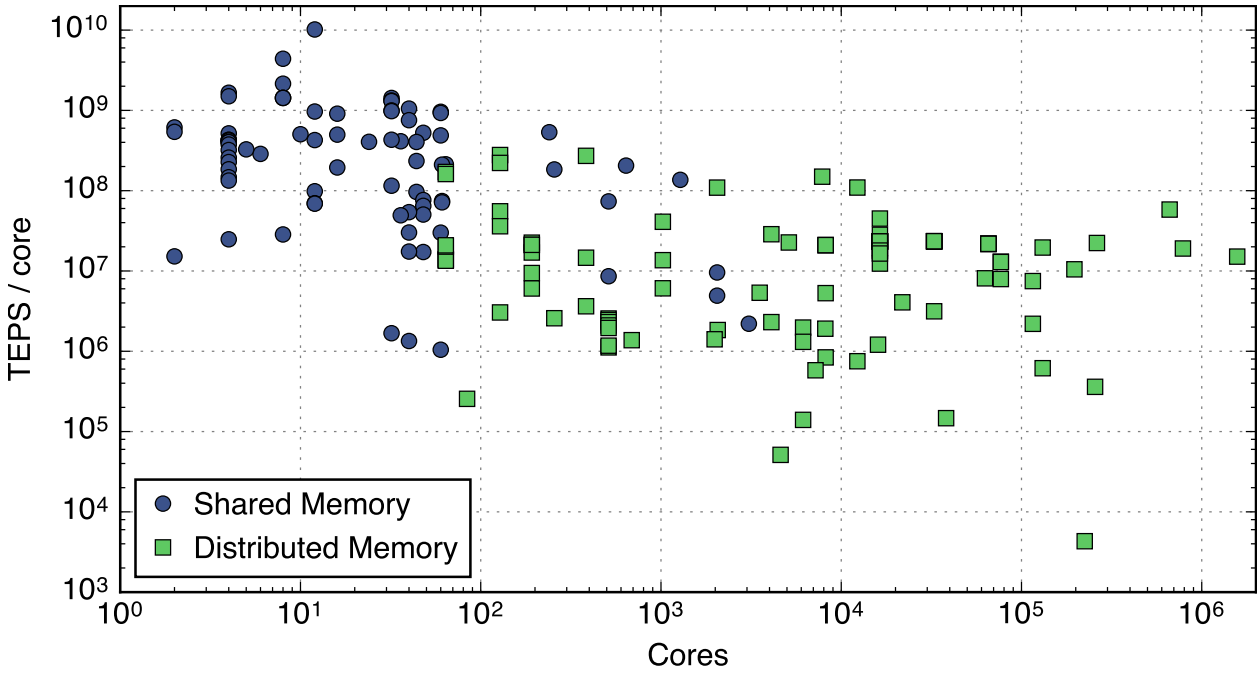

Figure 1.1: Performance per core versus core count for June 2015 Graph 500 rankings [66]. Performance measures traversed edges per second (TEPS) executing breadth-first search.

1.1 Focus on Shared Memory Multiprocessor Systems

We focus on single-node shared memory systems for this work. Graph processing research is ongoing for a variety of hardware platforms, such as clusters, semi-external memory systems, and even GPU-based systems, but many of these platforms are built from multiprocessor systems. Analyzing and improving multiprocessor systems first should immediately benefit the other platforms that are built from multiprocessors. Furthermore, we argue multiprocessor systems are well suited for many graph processing applications.

A large cluster is the typical tool for large computational problems, but graph processing frequently exacerbates the communication bottlenecks of large clusters. Since the interprocessor network of a distributed memory cluster is substantially slower than that of a shared memory multiprocessor system, clusters obtain substantially less performance per core for graph processing. For example, on Graph 500 [66], a world ranking of the fastest supercomputers for graph algorithms, the performance per core of distributed memory systems is often one to two orders-of-magnitude lower than that of shared memory systems (Figure 1.1). The high core count shared memory systems are implemented with interconnects similar to cluster interconnects, and their performance is similarly bottlenecked. This communication-bound behavior has led to surprising results, where a single Mac Mini operating on a large graph stored in an SSD is able to outperform a medium-sized cluster [92].

Due to the inefficiency of distributed graph processing, the primary reason to use a cluster for graph processing is if the data is too large to fit on a single node [105]. However, many interesting graph problems are not large enough to justify a cluster. For example, the entire Facebook friend graph can be held in only a few terabytes of uncompressed data [8], which can reside in a current high-end server's memory. Additionally, semi-external memory approaches that expand a server's capacity by using storage (flash or hard drives), are a much more economical means to handle larger problems. Multiprocessor systems have additional benefits, as they are easier to program than the other platforms, and their flexibility allows them to implement the most sophisticated optimized algorithms. For graph processing, multiprocessor systems are surprisingly capable systems that deliver robust performance.

1.2 Thesis Overview

In Chapter 2, we provide context for this work. We review our graph terminology to familiarize the reader with our notation. We also introduce the graphs we use and highlight their relevant topological properties. Finally, we survey graph processing research to identify the most commonly evaluated graph kernels that will constitute our workload for analysis.

Breadth-first search is a commonly used graph traversal within other algorithms, and in Chapter 3, we introduce our novel direction-optimizing breadth-first search algorithm. Our algorithm has been widely adopted, both by graph processing frameworks and within the Graph 500 competition. On low diameter graphs, our algorithm is able to avoid traversing a large fraction of the graph while still producing the correct result. In practice on real hardware, the speedup obtained by our algorithm is substantially less than the reduction in edges examined, which implies that the implementation of our algorithm is less efficient per traversed edge than our conventional baseline. This loss of efficiency motivates the rest of this work, as an algorithmic improvement resulted in substantial efficiency degradation on current hardware.

To characterize graph algorithms as a workload, we first need a workload to analyze. In Chapter 4, we introduce our GAP Benchmark Suite. We created the benchmark not only to create a workload for our own analysis, but also to help ameliorate the evaluation pitfalls we observe in graph processing research. With our benchmark suite, we hope to guide the community away from common evaluation mistakes such as using too few kernels, using too few input graphs, using input graphs that are too small, using only synthetically generated graphs, or inconsistent methodologies. In addition to a specification of our benchmark, we also provide a high-quality reference implementation for use as a baseline for evaluations.

In Chapter 5, we characterize the workload specified by our benchmark on real hardware. To identify hardware bottlenecks, we access our platform's performance counters to measure utilizations throughout the system. We use a simple model and synthetic microbenchmarks to establish how much memory bandwidth is available on our evaluation platform under various conditions. With this insight into the platform's available memory bandwidth, we observe that contrary to conventional wisdom, graph algorithms are typically not memory

bandwidth-bound. We find this low bandwidth utilization is due in large part to the cache hierarchy delivering a substantial number of cache hits.

To ease reasoning about graph algorithm performance, in Chapter 6 we introduce the Graph Algorithm Iron Law (GAIL). GAIL is a simple performance model that attributes performance differences to changes in algorithmic work, cache utility, or memory bandwidth utilization. Using GAIL, we perform a variety of case studies including analyzing our direction-optimizing breadth-first search algorithm. We determine our breadth-first search algorithm is less efficient per edge because it has worse cache locality and lower memory bandwidth utilization. In the chapter, we also argue how GAIL could be useful to a variety of graph researchers, as it can help weigh tradeoffs between algorithms, implementation decisions, and hardware platforms.

To reduce memory communication for graph algorithms, in Chapter 7 we introduce propagation blocking. Our technique is motivated by the observation that communication behavior for graph algorithms is different when accessing values associated with vertices versus accessing graph edges. Our propagation-blocking technique is designed to improve poor spatial locality when accessing vertex values. Despite performing more theoretical communication, our propagation-blocking technique achieves reductions in memory communication on real hardware.

In Chapter 8 we not only conclude this work by summarizing its contributions, but also suggest potential future work based on the results of this dissertation.

Chapter 2

Background on Graph Algorithms

In this chapter, we provide a background on graph processing in order to provide context for the rest of this work. We review the terminology and notation we use to describe graphs. We also introduce the input graphs we use for evaluations, and we highlight their relevant topological properties. Finally, we describe the graph kernels we use and our justification for selecting them based on their popularity in prior work.

2.1 The Graph Abstraction

The graph abstraction is a way to model the connections between objects and the nature of these connections can have many properties. More formally, a *graph* $G(V, E)$ is composed of a set of vertices V and a set of edges E. An *edge* (u, v) connects the two vertices u and v. A *directed edge* (u, v) represents a connection from u to v, while an *undirected edge* represents a bidirectional connection. If a graph is composed exclusively of undirected edges, it is an *undirected graph*, otherwise, it is a *directed graph*. The *degree* of a vertex is the number of edges connected to it. The degree of a graph is the average degree of all of its vertices ($|E|/|V|$). To capture the diversity of vertex degrees within a graph, the *degree distribution* is the distribution of degrees over the vertices within a graph.

The *distance* $d(u, v)$ between vertex u and vertex v is the fewest number of hops between u to v. The *depth* of a vertex is the distance between a vertex and a starting source vertex of a traversal. The *diameter* of a graph is the largest distance between any two vertices in the graph. More informally, diameter often refers to the largest distance that is typical when ignoring outliers. In this work, we use the 99^{th}-percentile distance to define our effective diameter.

Most graph algorithms operate on more than connectivity information and they associate metadata with vertices and edges. For example, many traversal algorithms associate a boolean value with each vertex indicating if it has been visited. A *weighted edge* (u, v, w) associates a numerical cost w with traversing the edge and a *weighted graph* is composed of weighted edges. The distance $d(u, v)$ between vertex u and vertex v in a weighted graph is

the sum of the edge weights of the minimum total weight path between u and v.

We use some notation to enable concise expressions in this work. We use n and m to refer to the number of vertices ($|V|$) and the number of edges ($|E|$) respectively. We use $k = m/n$ to represent a graph's average degree. The set of vertices directly reachable from vertex v (outgoing neighborhood) is $N^+(v)$ and the set of vertices with edges pointing to vertex v (incoming neighborhood) is $N^-(v)$. If the graph is undirected, both neighborhoods are the same $N^+(v) = N^-(v)$.

2.2 Topological Properties of Graphs

Graph applications are characterized not only by the algorithms used, but also by the structure of the graphs that make up their workload. In this work, we divide commonly used graphs into two broad categories named for their most emblematic members: *meshes* and *social networks* [16]. Meshes tend to be derived from physically spatial sources, such as road maps or the finite-element mesh of a simulated car body, so they can be relatively readily partitioned along the few original spatial dimensions. Due to their physical origin, they usually have a high diameter and a degree distribution that is both bounded and low.

Conversely, social networks come from non-spatial sources, and consequently are difficult to partition using any reasonable number of dimensions. Additionally, social networks have a low diameter ("small-world") and a power-law degree distribution ("scale-free"). In a *small-world graph*, most vertices are not neighbors of one another, but most vertices can be reached from every other by a small number of hops [160]. More quantitatively, a small-world graph can be recognized by a low diameter. A *scale-free graph* has a degree distribution that follows a power law, at least asymptotically [14]. The fraction of vertices in a scale-free graph having k connections to other vertices is $P(k) \sim k^{-\gamma}$, where γ is a parameter typically in the range $2 < \gamma < 3$. These two properties can be readily recognized in real life social networks, as typically only a few people have many friendships (scale-free), and a surprisingly low number of friendships connects any two people (small-world) [109].

Meshes are perhaps the most common mental model for graphs, since they are typically used in textbook figures. Unfortunately, they do not capture the challenges posed by the small-world and scale-free properties of social network topologies. The small-world property makes them difficult to partition (few cut edges relative to enclosed edges), while the scale-free property makes it difficult to load balance a parallel execution since there can be many orders of magnitude difference between the work for different vertices. Although the highest degree vertices are rare, their incident edges constitute a large fraction of the graph.

A graph's *sparsity* is determined by its average degree. If vertices are connected to nearly every other vertex, it is a *dense graph*. If vertices are connected to only a few other vertices, it is a *sparse graph*. There is no specific degree cutoff between sparse graphs and dense graphs, but for a graph of n vertices, a dense graph has $O(n^2)$ edges (the maximum) while a sparse graph has $O(n)$ edges. This work restricts itself to sparse graphs since they are used

in more emerging graph processing applications and because processing sparse graphs incurs greater communication inefficiencies.

2.3 Graphs in this Work

In this work, we use the diverse set of graphs listed in Table 2.1 to guide our investigations. In Chapter 5, we demonstrate that it is important to have a diverse set of graphs, as a graph's topology greatly impacts the workload's characteristics. To keep our work relevant, most of our graphs are generated from real-world datasets. We primarily focus on social network topologies, as they are more challenging than meshes and they are at the epicenter of the renewed interest in graph processing.

The real-world data we use to generate our graphs comes from a variety of sources. Real-world social network data is often difficult to obtain due to anonymity concerns, so we are grateful for the publicly available real-world data we are able to use. The graphs facebook, flickr, friendster, livejournal, orkut, and twitter all represent the links between users on their respective online communities. In addition to considering social relationships, we also consider professional relationships with the hollywood graph that represents film actors and links them if they have performed together. Using the data on academic publications provided by the Microsoft Academic Graph [149], we generate a graph of all paper citations and a graph of all coauthorships. We also include two graphs (web and webbase) generated from web crawls that represent hyperlinks between websites. All of the real-world graphs in this study except the road graph have the social network topology, as they have low effective diameters and a power-law degree distribution. To contrast with the social networks, we include the road graph as an example of a mesh topology with its high diameter, low average degree, and low maximum degree.

Even though our graph suite includes some of the largest publicly available real-world graphs, they do not fully use the memory capacity of our evaluation system. As is done in the Graph 500 benchmark, we synthetically generate arbitrarily large graphs to fill our memory capacity. We generate the kron graph from the Kronecker generator [94] and we generate the urand graph from a uniform random graph generator [58]. We use the kron graph to model a social network and we select the generator parameters to match those of Graph 500 [66]. The urand graph has no locality by design, however, it is also the most unrealistic and serves to discover lower bounds of performance. The urand graph generates the desired number of edges by giving each generated edge an equal probability of connecting any two vertices. Hence, in our urand graph, each vertex tends to be accessed roughly the same number of times, unlike social networks in which the scale-free property causes a few vertices to be accessed disproportionately often. Since urand is generated to be the same size as kron, contrasting the two graphs allows us to observe the impact of the scale-free property.

Graph	Description	Vertices (M)	Edges (M)	Degree	Diameter	Directed
Kron	Kronecker generator [66, 94]	134.2	2,111.6	15.7	5	
Urand	uniform random [58]	134.2	2,147.4	16.0	6	
Citations	academic citations [149]	49.8	949.6	19.0	12	✓
Coauthors	academic coauthorships [149]	119.9	646.9	5.4	10	
Facebook	social network [163]	3.0	28.3	9.2	6	
Flickr	social network [110]	1.8	22.6	12.2	12	✓
Friendster	social network [170]	124.8	1,806.1	14.5	7	
Hollywood	movie collaborations [27, 28, 47]	1.1	57.5	50.5	5	
LiveJournal	social network [110]	5.3	79.0	14.7	9	✓
Orkut	social network [110]	3.0	223.5	72.8	5	
Road	USA road network [50]	23.9	58.3	2.4	6,277	✓
Twitter	social network [91]	61.5	1,468.3	23.8	7	✓
Web	crawl of .sk domain [47]	50.6	1,949.4	38.5	13	✓
WebBase	2001 web crawl [47]	118.1	632.1	5.4	16	✓

Table 2.1: Graphs used for evaluation. All of the graphs come from real-world data except kron and urand. In subsequent chapters, we list the subsets of these graphs we use.

2.4 Graph Algorithms in this Work

In this work, we analyze six graph kernels, and we select them based on their popularity in graph processing framework research evaluations (Table 2.2). For more information on the frameworks we survey, please consult Appendix A. Broadly speaking, the kernels we use can be categorized as either traversal-centric or compute-centric. *Traversal-centric* kernels (BFS, SSSP, and BC) start from a given source vertex and perform their computation outwards from the source vertex. *Compute-centric* kernels (PR, CC, and TC) receive no source vertex as input and operate on the entire graph in parallel.

- **Breadth-First Search (BFS)**
  BFS is not even a graph algorithm, but is only a graph traversal order. It is commonly used within other graph algorithms, but BFS can be turned into an algorithm by tracking information related to the traversal. BFS traverses all vertices at one depth before moving onto the next depth. We analyze BFS thoroughly in Chapter 3.

- **Single-Source Shortest Paths (SSSP)**
  SSSP returns the distances of shortest paths from a given source vertex to every other reachable vertex. SSSP typically operates on weighted graphs, so the shortest paths consider the edge weights. In this work, we restrict ourselves to non-negative edge

Platform	Total	PageRank	Single-source Shortest Paths	Connected Components	Breadth-first Search	Triangle Counting	Betweenness Centrality	Conductance	n-Hop Queries	Alternating Least Squares	Strongly Connected Components	Approximate Diameter	Random Walk Sampling
Shared Memory	11	8	7	5	4	2	3	1	1	-	1	2	-
Semi-external Memory	7	5	2	5	6	3	1	1	1	2	1	-	-
Distributed Memory	29	21	13	13	7	5	4	2	3	2	2	1	3
GPU	4	4	3	2	4	-	1	-	-	-	-	-	-
HW Accelerator	3	3	3	-	1	-	-	2	-	-	-	-	-
Total	54	41	28	25	22	10	9	6	5	4	4	3	3
Percentage (%)		76	52	46	41	19	17	11	9	7	7	6	6

Table 2.2: Number of graph framework evaluations using each graph algorithm, categorized by hardware platform. Only 12 most commonly evaluated algorithms are shown. See Appendix A for more information on survey of graph processing frameworks.

weights. If the graph is unweighted, BFS can return the shortest paths since all edges will have the same unit weight.

In addition to the distances, some SSSP implementations also return the parent vertices along the shortest paths. In Section 6.3 we evaluate the tradeoffs in implementing delta-stepping [107], a pragmatic parallel SSSP algorithm.

- **PageRank (PR)**
  PageRank determines the "popularity" of vertices in a graph, and it was originally used to sort web search results [127]. PageRank determines the popularity of a vertex v not only by the number of vertices that point to v, but also the popularity of the vertices that point to v. More formally, the PageRank score (PR) for a vertex v with a damping factor d (0.85) is:

$$PR(v) = \frac{1-d}{|V|} + d \sum_{u \in N^-(v)} \frac{PR(u)}{|N^+(u)|}$$

  The above recurrence often results in cyclic dependencies, but as long as the graph is aperiodic, the scores will converge [22]. PageRank typically iterates until the scores

converge within a specified tolerance of a fixed point.

PageRank is a popular graph benchmark because it exposes many of the challenges of graph processing while still being simple enough to ease analysis and implementation. Correspondingly, there is considerable prior work on accelerating PageRank. A common optimization for PageRank exploits the differences in vertices' convergence rates by not processing vertices whose scores change too little relative to the previous iteration [96, 146]. Another common optimization is to apply the Gauss-Siedel method and update scores "in-place" by overwriting the previous score with the new score [90]. This optimization improves the convergence rate since the computation will often be operating on newer data.

- **Connected Components (CC)**
  The connected components algorithm labels the components of a graph. If there is a path between two vertices, those two vertices are *connected*. A *connected component* is a subgraph such that all of its vertices are connected to each other. A connected component is maximal, as any vertex that is connected to the component is part of the component.

  In a directed graph, connection relationships can be asymmetric and thus not commutative. For example, if there is a directed edge from u to v, u is connected to v, but v may not be connected to u. In a directed graph, connected components can either be strongly connected or weakly connected. A *strongly connected component* is a subgraph such that there is a directed path between every pair of its vertices. A *weakly connected component* is a connected component when ignoring edge directions, and a strongly connected component is also a weakly connected component. On a directed graph, the term "connected components" is ambiguous, but typically refers to weakly connected components.

  The connected components algorithm labels vertices such that all vertices in the same component get the same label. Vertices of zero degree are not connected to any other vertices and are thus their own components so they get their own labels.

- **Betweenness Centrality (BC)**
  Betweenness centrality is a metric that attempts to measure the importance of vertices within a graph. In particular, BC creates a score for each vertex that measures the fraction of shortest paths that pass through that vertex. More formally, if σ_{st} is the number of shortest paths between vertices s and t, and $\sigma_{st}(v)$ is the number of those shortest paths that pass through vertex v, the betweenness centrality score for v is:

$$BC(v) = \sum_{s,t \in V, s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

  BC can be computationally demanding as it requires computing *all* of the shortest paths between *all* pairs of vertices. In practice, this is often accomplished by executing

SSSP from every vertex as a source. Executing SSSP for every vertex is not only a great deal of shortest paths to compute, but can also consume a great deal of memory capacity to store all of the computed shortest paths. The Brandes algorithm greatly reduces memory requirements, as it is able to compact the critical information from a single SSSP execution into a single value per vertex [29]. The reduction in memory capacity requirements greatly improves the tractability of computing BC, however, to compute the exact scores the Brandes algorithm still needs to perform SSSP from every vertex. Many use cases of BC only care about the relative ordering of vertices, so approximations can be sufficient. The easiest way to approximate BC scores is to only perform SSSP from a subset of the vertices, and if these vertices are selected in a sufficiently random way, the approximation is reasonable [13]. The final BC scores are typically normalized to one. Since BC is often performed on unweighted graphs, the SSSP executions can be accomplished by using BFS traversals.

- **Triangle Counting (TC)**
  Triangle counting is used to measure the interconnectedness of a graph. A *clique* is a set of vertices that are all directly connected to each other. As the size of a clique grows, the number of edges it contains grows quadratically, so within real-world graphs, large cliques are rarely seen. A *triangle* is a clique of size 3, and due to its small size, it occurs often in nature. For example, in a friendship network, a triangle represents two friends having another friend in common. The presence of many triangles in a graph implies the graph has densely interconnected clusters, even if these clusters do not form perfect cliques.

  As triangle counting's name suggests, the algorithm counts the number of triangles in a graph. In some usage scenarios the list of triangles is desired, but in our usage, the output of TC is the total number of triangles in the graph. Triangles are invariant to permutation, so three interconnected vertices can only be counted as a triangle once. We do not consider edge directions when defining a triangle, so for TC, all edges can be treated as undirected.

2.5 Other Types of Graph Algorithms

Graphs can be used in many contexts and ways, and this work focuses on some of the more classic and widely used types of graphs and graph algorithms. In this section, we briefly highlight some of the other types of graph processing.

For some graph processing applications, there is so much graph data it cannot fit into the processing system's memory simultaneously. In *streaming graph processing*, the graph "streams" through the system and the algorithms must be able to operate on the edges as they flow through. Depending on the usage scenario, the graph could be read from storage [92] or a network source [151].

For some applications, the data the graph represents is changing in real-time, and so the graph topology also needs to change [54]. In *dynamic graph processing*, the graph topology is changing independently of the algorithm. This plasticity is in contrast to a graph algorithm that deliberately mutates the graph topology as part of its computation. Some sophisticated dynamic graph algorithms can update a prior solution by only computing on the parts of the graph that have changed [55].

The graph abstraction is fundamentally centered around edges, and an edge models a relation between two vertices. A *hypergraph* generalizes this relation, and a *hyperedge* can represent a relation between any number of vertices. Although a conventional graph can connect more than two vertices by using multiple edges, a hyperedge semantically conveys that those vertices are connected by the same relation. Hypergraphs can sometimes be challenging to implement efficiently, so they are often transformed into conventional bipartite graphs. Each hyperedge can be represented as a vertex with an outgoing edge for each of the hyperedge's connections. An interesting application of hypergraphs is partitioning conventional graphs [35].

Graph algorithms can also be expressed with linear algebra [87]. A graph can be represented as a matrix in which each non-zero represents a relation between the row and the column. Since many graphs of interest are sparse, their corresponding matrices are also sparse. Using the linear algebra abstraction not only provides great notational conciseness and expressivity, it can often allow for reusing optimized linear algebra libraries. Unfortunately, the parallelism expressed by linear algebra prevents some algorithmic optimizations such as our bottom-up search optimization presented in the next chapter [20].

Chapter 3

Direction-Optimizing Breadth-First Search

In this chapter, we present *direction-optimizing BFS*, an algorithmic innovation we use to motivate the rest of this work's focus on understanding graph algorithm performance. We not only demonstrate practical speedups using our direction-optimizing algorithm, we also provide insight into the topological properties that enable the drastic reduction in edges examined that enables our algorithm's speedup. The execution time speedup of our approach is typically less than the reduction in edges examined, implying our implementation is less efficient per edge examined. We further explain the architectural causes of this gap in Chapter 6.

3.1 Introduction

Breadth-First Search (BFS) is an important building block used in many graph algorithms, and it is commonly used to test for connectivity or compute the single-source shortest paths of unweighted graphs. Due to its simplicity and wide use, BFS is often used as a graph processing benchmark [66], but its low computational intensity often results in low performance. To accelerate BFS, there has been significant prior work to change the algorithm and data structures, in some cases by adding additional computational work, to increase locality and boost overall performance [2, 33, 77, 173]. However, none of these previous schemes attempt to *reduce* the amount of computational work (number of edges examined).

We present the *direction-optimizing BFS*, a hybrid algorithm that combines the conventional top-down approach with a novel bottom-up approach [16]. By examining substantially fewer edges, the new algorithm obtains speedups of 1.5–6.7× on a suite of synthetic and real-world social network graphs. In the top-down approach, vertices in the active frontier search for an unvisited child, while in our new bottom-up approach, unvisited vertices search for a parent in the active frontier. In general, the bottom-up approach will yield speedups when the active frontier is a substantial fraction of the graph, which commonly occurs in

small-world graphs such as social networks.

The bottom-up approach is not always advantageous, so we combine it with the conventional top-down approach, and use a simple heuristic to dynamically select the appropriate approach to use at each step of a BFS. We show that our dynamic on-line heuristic achieves performance within 15% of the optimum possible using an off-line oracle. An early version of the direction-optimizing BFS algorithm [15] running on a stock quad-socket Intel server was ranked 17^{th} in the Graph500 November 2011 rankings [66], achieving the fastest single-node implementation and the highest per-core processing rate, and outperforming specialized architectures and clusters with more than 150 sockets. In the following Graph500 competitions, the direction-optimizing algorithm was adopted by most of the top finishers [38, 59]. Additionally, the direction-optimizing algorithm has also been widely adopted in graph frameworks and benchmark suites [18, 31, 61, 118, 122, 146, 158].

3.2 Conventional Top-Down BFS

Top-down BFS starts from a source vertex, expands the frontier outwards during each step, and visits all of the vertices at the same depth before visiting any at the next depth (Figure 3.1). During a step of the conventional top-down approach (Figure 3.2), each vertex checks all of its neighbors to see if any of them are unvisited. Each previously unvisited neighbor is added to the next frontier and marked as visited by setting its parent variable. This algorithm yields the BFS tree, which spans the connected component containing the source vertex. Other variants of BFS may record other attributes instead of the parent (predecessor) at each vertex in the BFS tree, such as a simple boolean variable that marks whether it was visited, or an integer representing its depth in the tree.

The behavior of BFS on social network topologies follows directly from their defining properties: small-world and scale-free. Because social networks are small-world graphs, they have a low effective diameter, which reduces the number of steps required for a BFS traversal, which in turn causes a large fraction of the vertices to be visited during each step. The scale-free property requires some vertices to have much higher degrees than average, allowing the frontier growth rate to outpace the average degree. As a result of these two properties, the size of the frontier ramps up and down exponentially during a BFS traversal of a social network. Even if a social network graph has hundreds of millions of vertices, the vast majority will be reached in the first few steps.

The majority of the computational work in BFS is finding unvisited neighbors of the frontier. Unvisited neighbors are found by "checking" the edges of the frontier to see if the endpoint has been visited. The total number of edge checks for an entire BFS with the conventional top-down approach is equal to the number of edges in the connected component containing the source vertex, as on each step every edge in the frontier is checked.

Figure 3.3 shows a breakdown of the result of each edge check for each step during a conventional parallel top-down BFS traversal on the kron graph (synthetic social network graph used for the Graph500 benchmark [66]). The middle steps (depths 2 and 3) consume

```
breadth-first-search(graph, source)
    frontier ← {source}
    next ← {}
    parents ← [-1,-1,...-1]
    while frontier ≠ {} do
      top-down-step(graph, frontier, next, parents)
      frontier ← next
      next ← {}
    end while
    return parents
```

Figure 3.1: Conventional BFS algorithm

```
top-down-step(graph, frontier, next, parents)
    for v ∈ frontier do
      for u ∈ neighbors[v] do
        if parents[u] = -1 then
          parents[u] ← v
          next ← next ∪ {u}
        end if
      end for
    end for
```

Figure 3.2: Single step of top-down approach. Vertex v unvisited if parents$[v] = -1$.

the vast majority of the runtime, which is unsurprising since the frontier is then at its largest size, requiring many more edges to be examined. During these steps, most attempts to become the parent of a neighbor are unsuccessful because the neighbor has already been visited. These visited neighbors can be broken down into three different categories based on their depth relative to the candidate parent: *potential parent*, *peer*, and *unavailable child*. A potential parent is any neighbor at depth $d - 1$ of a vertex at depth d. A peer is any neighbor at the same depth. An unavailable child is any neighbor at depth $d + 1$ of a vertex at depth d, but at the time of examination it has already been claimed by another vertex at depth d. Successful checks result in a *claimed child*. Figure 3.4 demonstrates these neighbor categorizations for a trivial example search. Figure 3.3 shows most of the edge checks are unsuccessful (not claimed children) and thus represent redundant work, since a vertex in a correct BFS tree only needs one parent.

The progression of neighbor types in Figure 3.3 is typical among the social networks examined. During the first few steps, the percentage of claimed children is high, as the vast majority of the graph is unexplored, enabling most edge checks to succeed. During the next few steps, the percentage of unavailable children rises, which is unsurprising since the frontier has grown larger, and multiple potential parents fight over children. As the frontier

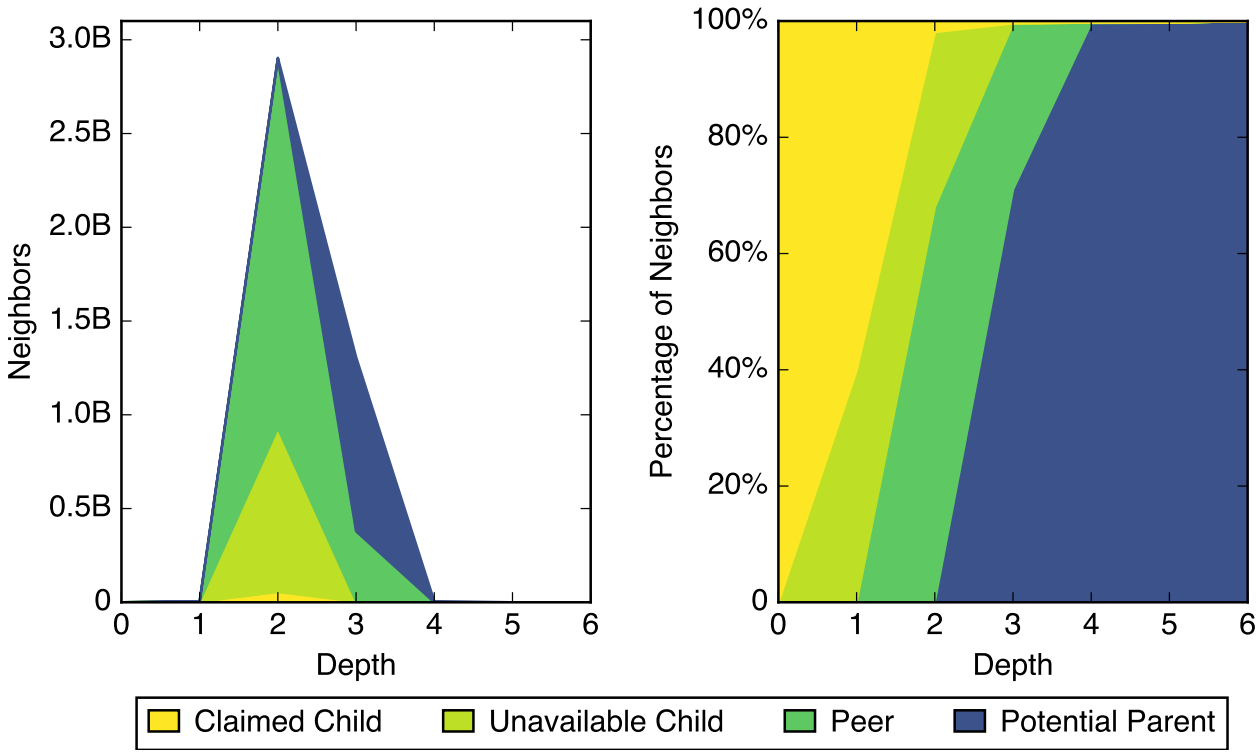

Figure 3.3: Classification of neighbors of frontier in terms of absolute totals (left) or normalized per depth (right) for a BFS traversal on kron graph (synthetically-generated social network of 128M vertices and 2B undirected edges). A single undirected edge can result in two neighbors of the frontier if both endpoints are in the frontier.

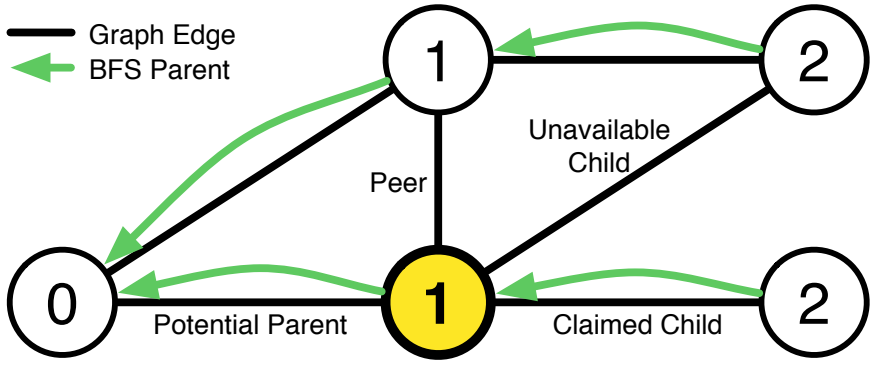

Figure 3.4: Categorization of the neighbors of the highlighted vertex for an example BFS traversal. The vertices are labelled with their depths from the BFS, and the arrows point from child to parent from the BFS.

reaches its largest size, the percentage of peer edges dominates. Since the frontier is such a large fraction of the graph, many edges must connect vertices within the frontier. As the frontier size rapidly decreases after its apex, the percentage of potential parents rises since such a large fraction of edges were in the previous step's frontier.

If a traversal was guided by an omniscent oracle, the theoretical minimum number of edges that need to be examined in the best case is one less than the number of vertices in the BFS tree, since that is how many edges are required to connect them. For the example in Figure 3.3, only 63,036,109 vertices are in the BFS tree, so at least 63,036,108 edges need to be considered, which is about $\frac{1}{67}^{th}$ of all the edge examinations that would happen during a top-down traversal. This factor of 67 is substantially larger than the input degree of 16 for two reasons. First, the input degree is for undirected edges, but during a top-down search each edge will be checked from both endpoints, doubling the number of examinations. Secondly, there are a large number of vertices of zero degree, which misleadingly decreases the average degree. Since for this example the classic top-down approach will examine 67× more edges than necessary, there is clearly substantial room for improvement by checking fewer edges, although in the worst case, every edge might still need to be checked.

3.3 Bottom-Up BFS

When the frontier is large, there exists an opportunity to perform the BFS traversal more efficiently by searching in the reverse direction, that is, *bottom-up*. Note that at depth 3 in the example (Figure 3.3), there are many potential parents, but those same edges mostly result in unavailable children when examined in the opposite direction during the previous step at depth 2. We can exploit this phenomenon to reduce the total number of edges examined. Instead of each vertex in the frontier attempting to become the parent of *all* of its neighbors, each unvisited vertex attempts to find *any* parent among its neighbors. A neighbor can be a parent of an unvisited vertex if the neighbor is a member of the frontier. The advantage of this approach is that once a vertex has found a parent, it does not need to check the rest of its neighbors. Figure 3.3 demonstrates that for some steps, a substantial fraction of neighbors are valid parents, so the probability of not needing to check every edge is high. Figure 3.5 shows the algorithm for a single step of this bottom-up approach.

The bottom-up approach also removes the need for some atomic operations in a parallel implementation. In the top-down approach, there could be multiple parallel writers to the same child, so atomic operations are needed to ensure mutual exclusion. With the bottom-up approach, only the child writes to itself, removing any contention. This advantage, along with the potential reduction of edges checked, comes at the price of serializing the work for any one vertex, but there is still massive parallelism between the work for different vertices.

If the graph is undirected, performing the bottom-up approach requires no modification to the graph data structures as both directions are already represented. If the graph is directed, the bottom-up step will require the inverse graph, which if not already available, can nearly double the graph's memory footprint. The bottom-up approach is advantageous

```
bottom-up-step(graph, frontier, next, parents)
    for v ∈ vertices do
      if parents[v] = -1 then
        for u ∈ neighbors[v] do
          if u ∈ frontier then
            parents[v] ← u
            next ← next ∪ {v}
            break
          end if
        end for
      end if
    end for
```

Figure 3.5: Single step of bottom-up approach

when a large fraction of the vertices are in the frontier, but will result in more work if the frontier is small. Hence, an efficient BFS implementation must combine both the top-down and bottom-up approaches.

3.4 Direction-Optimizing BFS Algorithm

The top-down approach and the bottom-up approach are complementary, since when the frontier is its largest, the bottom-up approach will be at its best whereas the top-down approach will be at its worst, and vice versa. The runtime for either the top-down approach or the bottom-up approach is roughly proportional to the number of edges they examine. The top-down approach will examine every edge emanating from the frontier while the bottom-up approach could examine every edge attached to an unvisited vertex, but hopefully fewer. Figure 3.6 illustrates this behavior by showing the time per step for each approach using the same example search as in Section 3.2. As the size of the frontier ramps up, the time per step of the top-down approach rises correspondingly to track the number of edges in the frontier, but the time per step for the bottom-up approach drops.

Our hybrid algorithm uses the top-down approach for steps when the frontier is small and the bottom-up approach for steps when the frontier is large. We begin each search with the top-down approach and continue until the frontier becomes too large, at which point we switch to the bottom-up approach. Although it is difficult to tell from Figure 3.6, it is usually worthwhile to switch back to the top-down approach for the final steps. During some searches there can be a long tail, and edges that are not in the connected component continue to consume runtime using the bottom-up approach. Since we perform the BFS traversal in whichever direction will be less work, we name it *direction-optimizing BFS*.

The step when transitioning from the top-down approach to the bottom-up approach provides an enormous opportunity to skip work, since all of the edge checks that would

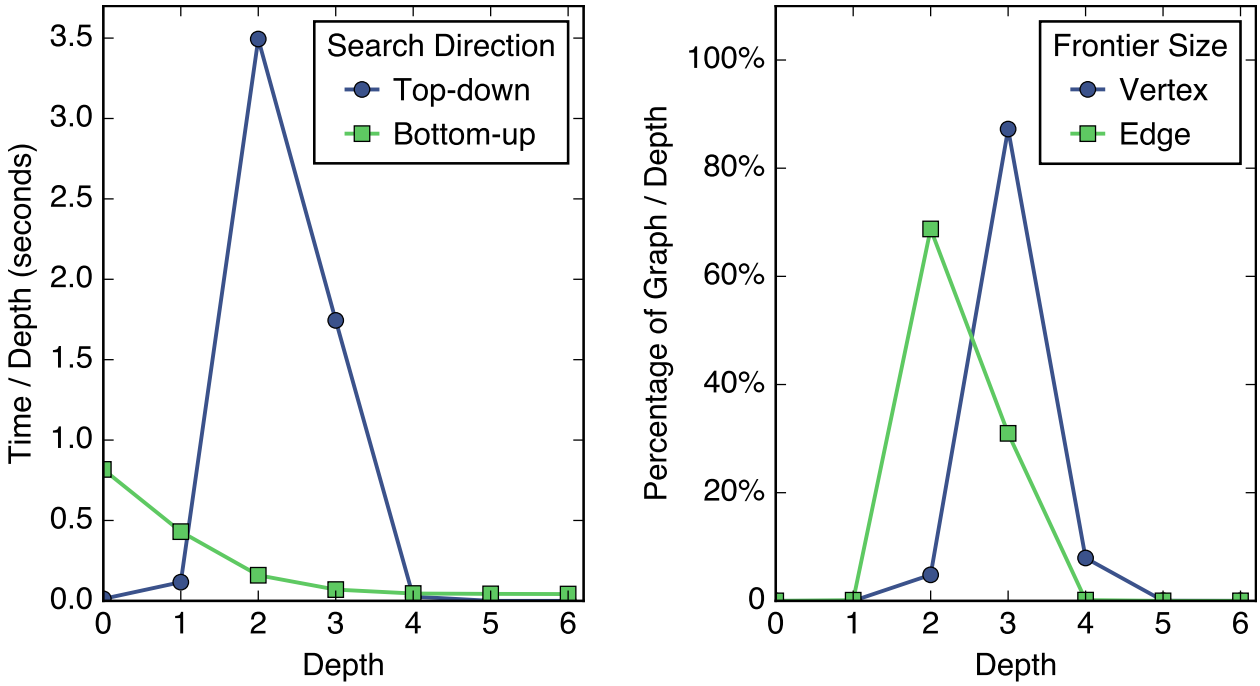

Figure 3.6: Example search on kron graph (synthetically-generated social network of 128M vertices and 2B undirected edges) on IVB platform from same source vertex as Figure 3.3. The left subplot is the time per depth if done top-down or bottom-up and the right subplot is the size of the frontier per depth in terms of either vertices or edges. The time for the top-down approach closely tracks the number of edges in the frontier, and depths 2 and 3 constitute the majority of the runtime since they constitute the majority of the edges.

happen during that top-down step (all of the edges in the frontier) can be skipped. For this reason, the optimum point to switch is typically when the number of edges in the frontier is at its largest. This is fortuitous, since the first bottom-up step will probably benefit from a high percentage of valid parents, as Figure 3.3 shows.

To control the hybrid approach, we use a dynamic heuristic based on: the number of edges to check from the frontier (m_f), the number of vertices in the frontier (n_f), and the number of edges to check from unexplored vertices (m_u). These metrics are efficient to compute, since they only require: summing the degrees of all vertices in the frontier, counting the number of vertices added to the frontier, or counting how many edges have been checked. Note that with an undirected graph, m_f or m_u might be counting some edges twice since each edge will be explored from both ends. Figure 3.7 shows the overall control heuristic, which compares m_f and n_f against two thresholds parameterized by α and β.

Since a step of the top-down approach will always check m_f edges, and the bottom-up approach will check at most m_u edges, if m_u is ever less than m_f, there is a guarantee that switching to the bottom-up approach at that step will check fewer edges. Since m_u is an overly pessimistic upper-bound for the number of edges the bottom-up approach will check,

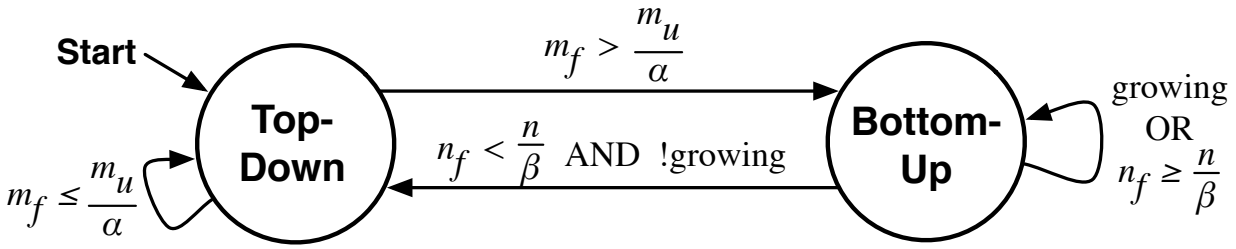

Figure 3.7: Finite-state machine to control the direction-optimizing algorithm. When transitioning between approaches, the frontier must be converted from a queue (top-down) to a bitmap (bottom-up) or vice versa. The *growing* condition indicates the number of active vertices in the frontier (n_f) increased relative to the last depth.

we use a tuning parameter α. This results in the condition for switching from top-down to bottom-up:

$$m_f > \frac{m_u}{\alpha}$$

Switching back to the top-down approach at the end of the traversal should occur when the frontier is small and there is no longer benefit to use the bottom-up approach. In addition to the overhead of checking edges outside the main connected component, the bottom-up approach becomes less efficient with a small frontier because it scans through all of the vertices to find unvisited vertices. While the frontier is still growing, we stay with the bottom-up approach, but once the frontier becomes too small, we switch back to the top-down approach and we use a heuristic parameterized by β:

$$n_f < \frac{n}{\beta}$$

3.5 Evaluation

Methodology

This section evaluates the performance of the direction-optimizing BFS approach empirically. Chapter 6 examines direction-optimizing BFS performance more deeply, and by leveraging insights from our workload characterization (Chapter 5), explains the processor architecture's impact on the algorithm. We elide formal worst-case complexity analysis for our algorithm, as the bounds will be too large to be informative. The number of edges examined by our algorithm is topology-dependent, and so the bounds will be dependent on how specific the model is. In practice, the direction-optimizing BFS is substantially faster for low-diameter graphs because it examines fewer edges.

We evaluate the performance of our hybrid algorithm using the low-diameter graphs in Table 3.1 (a subset of the graphs in Table 2.1). Our suite of test graphs includes both

Graph	Description	Vertices (M)	Edges (M)	Degree	Diameter	Directed
Kron	Kronecker generator [66, 94]	134.2	2,111.6	15.7	5	
Urand	uniform random [58]	134.2	2,147.4	16.0	6	
Citations	academic citations [149]	49.8	949.6	19.0	12	✓
Coauthors	academic coauthorships [149]	119.9	646.9	5.4	10	
Facebook	social network [163]	3.0	28.3	9.2	6	
Flickr	social network [110]	1.8	22.6	12.2	12	✓
Friendster	social network [170]	124.8	1,806.1	14.5	7	
Hollywood	movie collaborations [27, 28, 47]	1.1	57.5	50.5	5	
LiveJournal	social network [110]	5.3	79.0	14.7	9	✓
Orkut	social network [110]	3.0	223.5	72.8	5	
Twitter	social network [91]	61.5	1,468.3	23.8	7	✓
Web	crawl of .sk domain [47]	50.6	1,949.4	38.5	13	✓

Table 3.1: Low-diameter graphs used for evaluation. All of the graphs come from real-world data except kron and urand.

Architecture	Ivy Bridge EP
Intel Model	E5-2667 v2
Released	Q3 2013
Clock rate	3.3 GHz
# Sockets	2
Cores/socket	8
Threads/core	2
LLC/socket	25 MB
DRAM Capacity	128 GB

Table 3.2: IVB system specifications

synthetic (kron and urand) graphs as well as real social networks. The synthetic graph generators as well as their parameters are selected to match the Graph500 Competition [66] (Kronecker (A,B,C) = (0.57,0.19,0.19)). The real social networks are taken from a variety of web crawls [26–28, 47, 91, 110, 149, 161, 163]. A deeper discussion of the input graphs and their topological properties can be found in Section 2.3. For direction-optimizing BFS, the small-world property and the scale-free property are the most relevant topological properties to understand the algorithm's performance.

For our performance results, we use a dual-socket Intel Ivy Bridge server (IVB) that is representative of a compute node for a cluster (Table 3.2). We disable Turbo Boost to obtain consistent performance results. We operate on graphs stored in memory in Compressed Sparse Row (CSR) format after removing duplicate edges and self-loops. For each plotted data point, we perform BFS 64 times from pseudo-randomly selected non-zero degree vertices and average the results. During the BFS, we record the predecessor (parent) in the traversal for each visited vertex. The time for each search includes the time to allocate and initialize search-related data structures (including the `parents` array).

Implementations

For consistency, we perform all of our evaluation using implementations derived from the same C++/OpenMP codebase, and our optimized direction-optimizing implementation is publicly available as part of the GAP benchmark suite [60]. We incorporate best implementation practices, often exemplified by the original Graph500 `omp-csr` reference code [66]. Our top-down implementations represent the frontier with a shared queue that has thread-local buffers to prevent false sharing. For our optimized baseline, like Agarwal et al., we add a bitmap to track which vertices have already been visited [2]. This bitmap helps to reduce the number of costly random memory accesses to off-chip DRAM caused by checking if neighbors have been visited. By checking the bitmap first, many of these off-chip memory accesses can be avoided because the bitmap is often small enough to fit in the last-level cache.

For each search direction within our direction-optimizing implementation, we use a frontier representation specialized to the needs of each search direction's implementation. When switching search directions, the frontier representation must be appropriately converted, but the conversion should happen when the frontier is small, so the conversion cost is typically far less than the penalty of using the wrong data structure. The frontier is essentially a set, and based on the size of the frontier, it is more memory efficient to store it sparsely with an unordered queue or densely with a bitmap. The bottom-up approach also leverages the frontier bitmap to obtain a constant-time test for whether a particular vertex is in the frontier. In addition to saving memory, the top-down approach uses a frontier queue instead of a frontier bitmap since it is more efficient to iterate over when the frontier is small.

For the top-down implementation included within our direction-optimizing implementation, we perform an additional optimization to reduce the amount of time spent calculating how many edges exit the current frontier. The number of edges in the frontier is used in the top-down steps to determine if the frontier is large enough to justify switching to the bottom-up approach. Computing this total can have many irregular accesses, as it is summing the degrees of an unordered list of vertices. Our optimization is to add a step during initialization before the search that stores the degree of each vertex in the parent array as a negative number and this takes little time as the vertices can be processed in order with high spatial locality. The previous convention of -1 representing an unvisited vertex is now revised to be any negative number represents an unvisited vertex. With this new encoding,

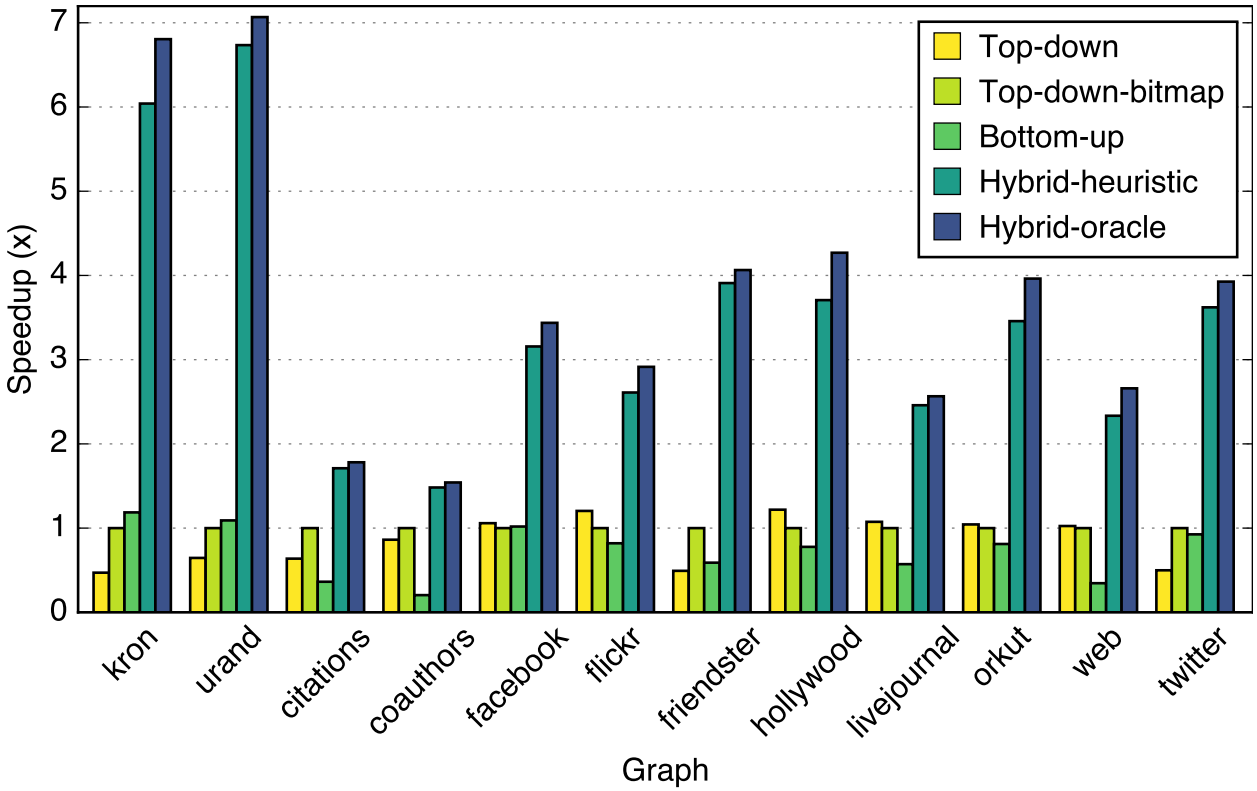

Figure 3.8: Speedups on IVB relative to top-down-bitmap.

the degree of a newly reached vertex is already known because the parent array was just checked to see if that vertex was unvisited. This optimization obviates the time for totaling the degrees of the frontier because it can now be done during the search. For low-diameter graphs, this yields a modest performance improvement, but for high-diameter graphs that will only use the top-down approach, this can yield a speedup of nearly 2×.

Comparing Algorithms

We first compare our own implementations executing on IVB for a range of low-diameter graphs (Figure 3.8). We use the following five implementations that leverage the techniques and optimizations described in the previous subsection:

Top-down is a standard parallel queue-based top-down BFS, very similar to the original Graph500 reference code.

Top-down-bitmap improves upon *top-down* by using the optimization of a bitmap to track previously visited vertices.

Bottom-up performs the bottom-up approach for all depths.

Hybrid-heuristic is our direction-optimizing implementation using our online control heuristic with $\alpha = 15$ and $\beta = 18$, and in the next subsection we describe how we tune α and β for IVB.

Hybrid-oracle repeatedly runs the direction-optimizing algorithm trying all possible transition points and then reports the best-performing to represent an oracle-controlled direction-optimizing implementation.

In Figure 3.8, we normalize execution time speedups to top-down-bitmap since it is representative of the current state-of-the-art. The use of a bitmap to track previously visited vertices typically delivers a substantial speedup over the standard top-down implementation. The direction-optimizing approach (hybrid implementations) provides large speedups across all of the graphs, with an average speedup of 3.4× and a speedup no lower than 1.5×. The on-line heuristic (hybrid-heuristic) often obtains performance within 10% of the oracle (hybrid-oracle), indicating that most of the time the heuristic chooses the correct step for the transition and the computational cost of the heuristic is reasonable. The pure bottom-up approach yields only modest speedups or even slowdowns, which highlights the importance of combining it with the top-down approach for the direction-optimizing approach.

Tuning α and β

For our direction-optimizing implementation (*hybrid-heuristic*) measured above, we tune the values of α and β. We first tune α before tuning β, as it has the greatest impact. Sweeping α across a wide range demonstrates that once α is sufficiently large (>10), BFS performance for many graphs is relatively insensitive to its value (Figure 3.9). This is because the frontier grows so rapidly that small changes in the transition threshold do not change the step at which the switch occurs. When trying to pick the best value for the suite, we select $\alpha = 15$ since it maximizes the average and minimum. Note that even if a less-than-optimal α is selected, the *hybrid-heuristic* algorithm still executes within 15–20% of its peak performance on most graphs. Additionally, a potential loss in performance due to a mistuned α is still far smaller than the speedup provided by direction-optimizing BFS (Figure 3.8).

Tuning β is less important than tuning α. We select $\beta = 18$, as this works well for the majority of the graphs (Figure 3.10). The value of β has a smaller impact on overall performance because the majority of the runtime is taken by the middle steps when the frontier is at its largest, even when those steps are accelerated by the bottom-up approach. We explore a much larger range for β since it needs to change by orders of magnitude in order to have an impact on which step the heuristic will switch. On IVB, setting β to 15 to make it equal to α does not substantially degrade performance, but in general there is no guarantee that α should equal β. For greater generality, we allow for the α and β thresholds to be set independently.

Although there is some variation between graphs for tuning the α and β parameters (Figure 3.9 & Figure 3.10), the performance of the top-down implementation relative to the

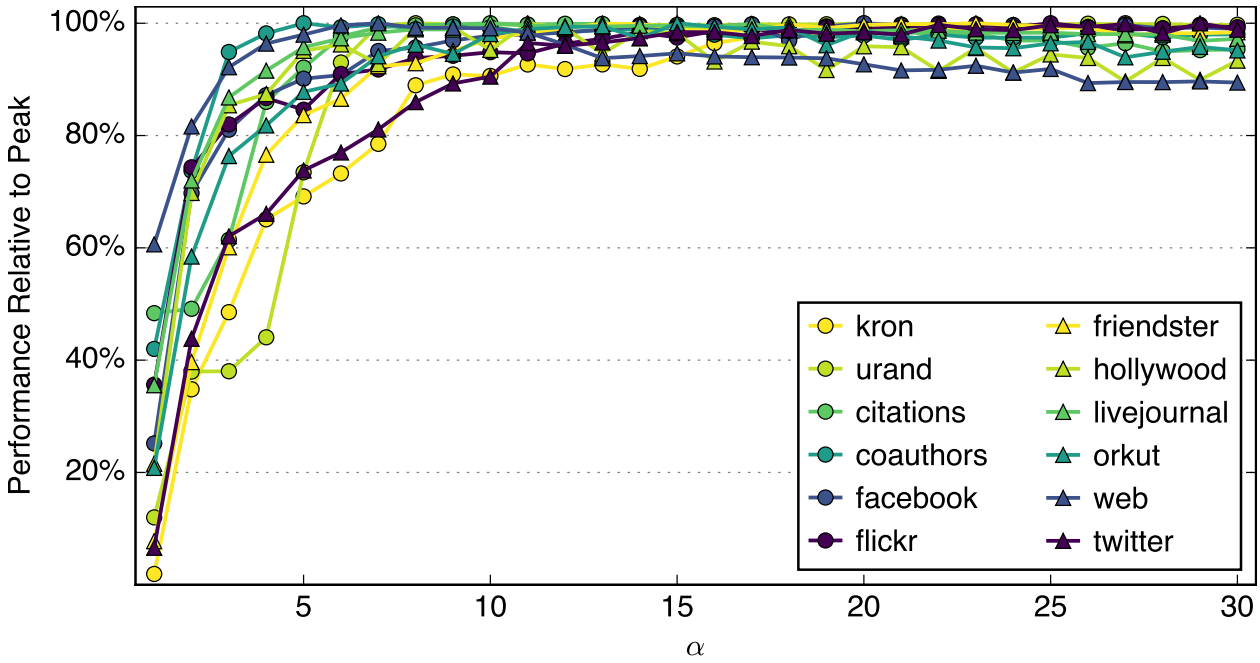

Figure 3.9: Performance of hybrid-heuristic on each graph relative to its best performance on that graph for the range of α examined. We select $\alpha = 15$.

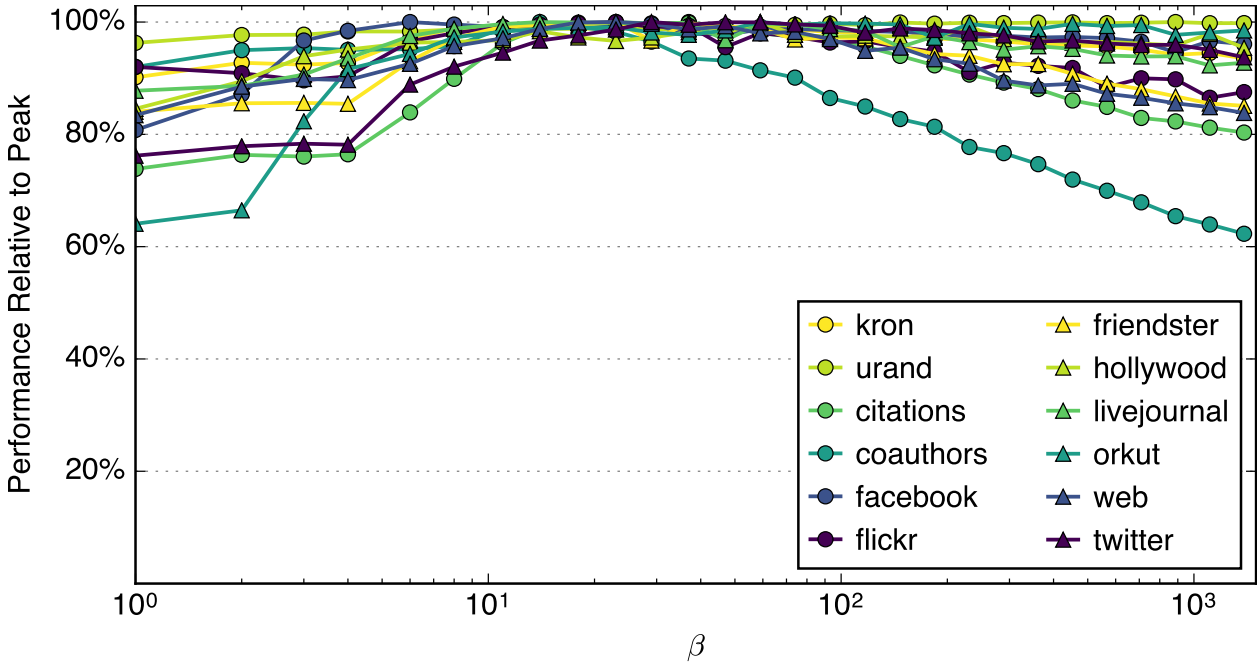

Figure 3.10: Performance of hybrid-heuristic on each graph relative to its best performance on that graph for the range of β examined. We select $\beta = 18$.

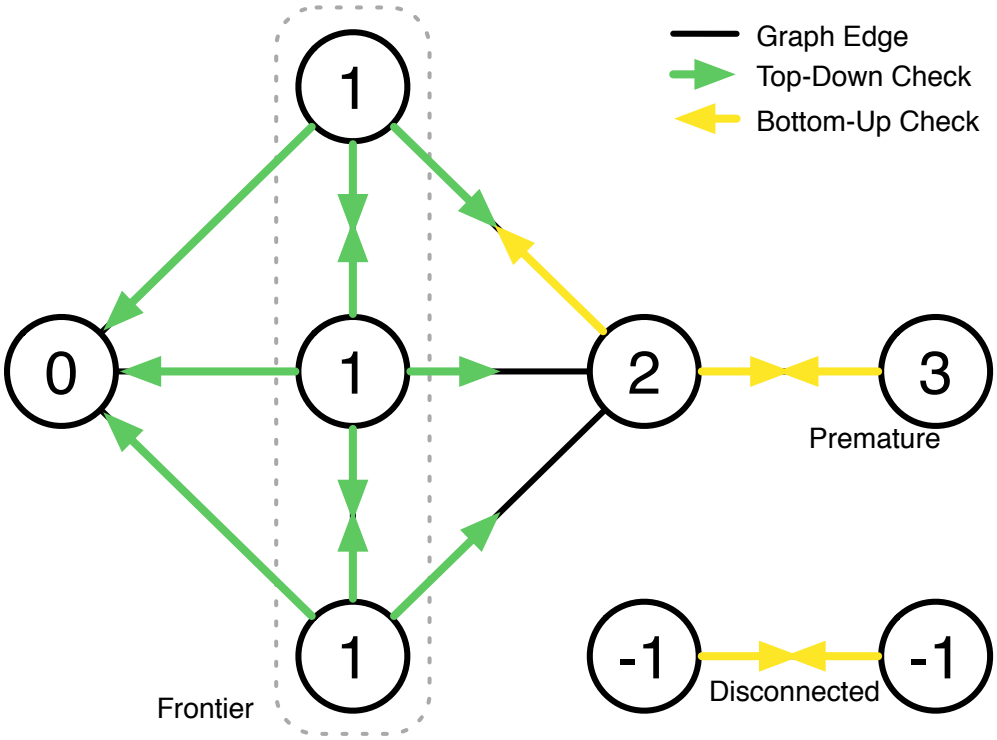

Figure 3.11: Comparison of edges examined by top-down and bottom-up on example graph when frontier is at depth 1. Vertices labelled with their final depth.

performance of the bottom-up implementation is the primary determinant of performance. For this reason, tuning the parameters can be done once for the platform, and should deliver reasonable performance for a range of input graphs.

Explaining the Performance Improvement

The speedup of the direction-optimizing approach demonstrated in Figure 3.8 is due to the reduction in edges examined. In a classical top-down BFS, every edge in the connected component containing the starting vertex will be checked, and for an undirected graph, each edge will be checked in both directions. The bottom-up approach skips checking some edges in two ways. First, once an unvisited vertex finds a parent, it can immediately stop examining its neighbors (early termination). Secondly, the step that transitions from the top-down approach to the bottom-up approach will skip edges on the transition itself.

A transition to the bottom-up approach at depth d skips all of the peer, potential parent, and unavailable child edges at depth d. These edge types can constitute the vast majority of the edges at a transition depth, as depth 2 of the example in Figure 3.3 exemplifies. The only edges from depth d that are not skipped by the transition are effectively the claimed child edges, and these edge are the requisite inter-depth edges between depths d and $d+1$,

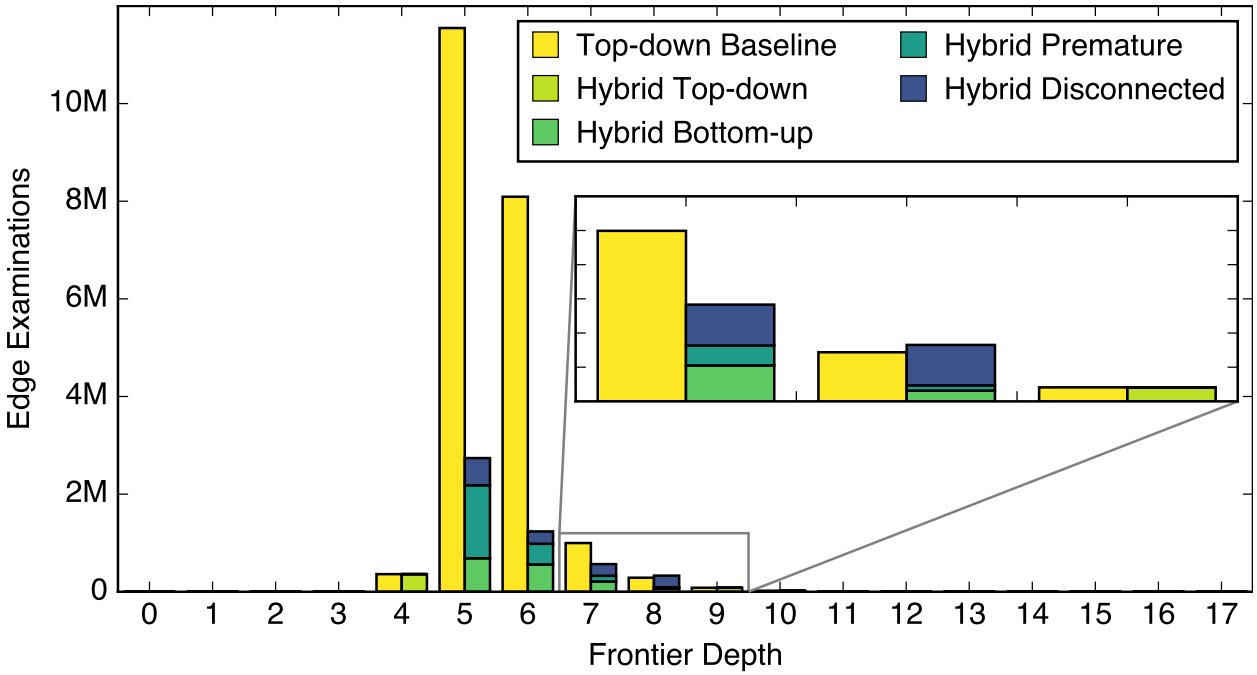

Figure 3.12: Types of edge examinations for top-down versus direction-optimizing for a single search on flickr graph. The direction-optimizing approach switches to the bottom-up direction at depth 5 and back to the top-down direction at depth 9. Due to extremely small frontier sizes, depths 0–4 and 9–17 are difficult to distinguish, however, for these depths the direction-optimizing approach traverses in the top-down direction so the edge examinations are equivalent to the top-down baseline.

as every vertex in the frontier at depth $d + 1$ must have exactly one parent at depth d.

To concretely demonstrate the savings in edge examinations from transitioning to the bottom-up approach, Figure 3.11 shows a simple example. With the frontier currently at depth 1, performing the traversal in the top-down direction requires examining every edge outgoing from a vertex at depth 1. By contrast, performing the traversal in the bottom-up direction possibly examines every edge incoming to a vertex of depth 2 or greater (if it is reached). Within the bottom-up traversal, the vertex at depth 2 is able to terminate its search for a parent early once it finds a neighbor of depth 1. Because the bottom-up search considers all unvisited vertices, some of these vertices are not adjacent to the frontier and thus result in wasteful edge examinations because there is no chance of them succeeding in finding a parent. The vertex at depth 3 is simply too far from the frontier, so we refer to its edge examination as *premature*. We refer to the edge examinations to vertices outside the main connected component (depth -1) as *disconnected*. In this simple example when the frontier is at depth 1, despite the wasteful examinations, the bottom-up approach examines fewer edges that the top-down approach (5 instead of 10).

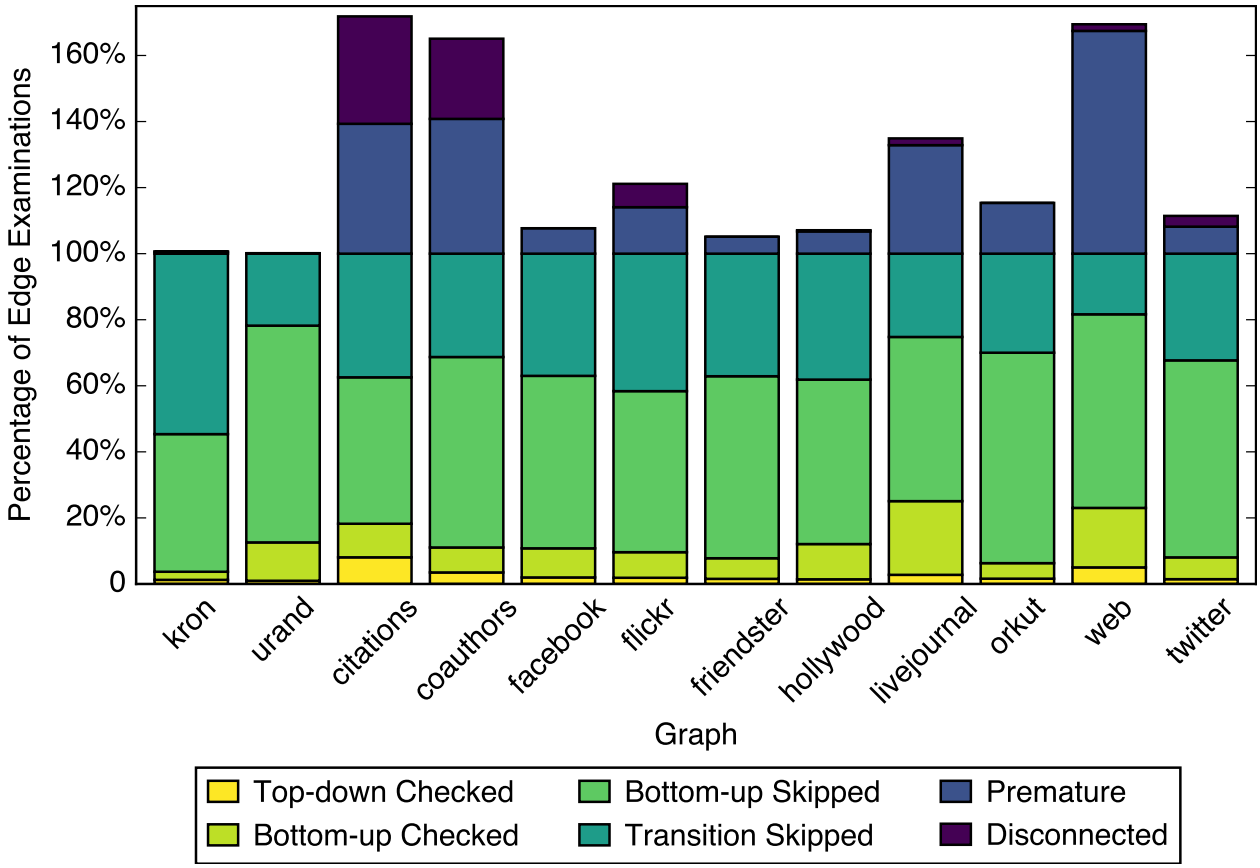

Figure 3.13: Categorization of edge examinations by hybrid-heuristic

Figure 3.12 compares the number and types of edges examined by the direction-optimizing approach versus the top-down approach for a single traversal of the flickr graph. We select the flickr graph because its traversal is less efficient for the direction-optimizing approach (only 2.6× speedup from a 3.2× reduction in edges examined), making its overheads more visible. For depths 0–4, both implementations examine the same number of edges since the hybrid implementation is still traversing in the top-down direction. At depth 5 the hybrid transitions to the bottom-up direction, and it enjoys a large reduction in edges examined, not only from early termination but also from the transition itself. At depth 5, the premature edge examinations are substantial, but as the traversal progresses, there are fewer unvisited vertices and thus fewer premature edges to examine. The overhead of disconnected examinations becomes more substantial as the frontier decreases in size, and at depth 8 it even causes the hybrid implementation to examine more edges than the top-down implementation. However, since the frontier dramatically reduces in size for the later depths, these penalties are drowned out, and the direction-optimizing approach is able to deliver a substantial reduction in overall edges examined.

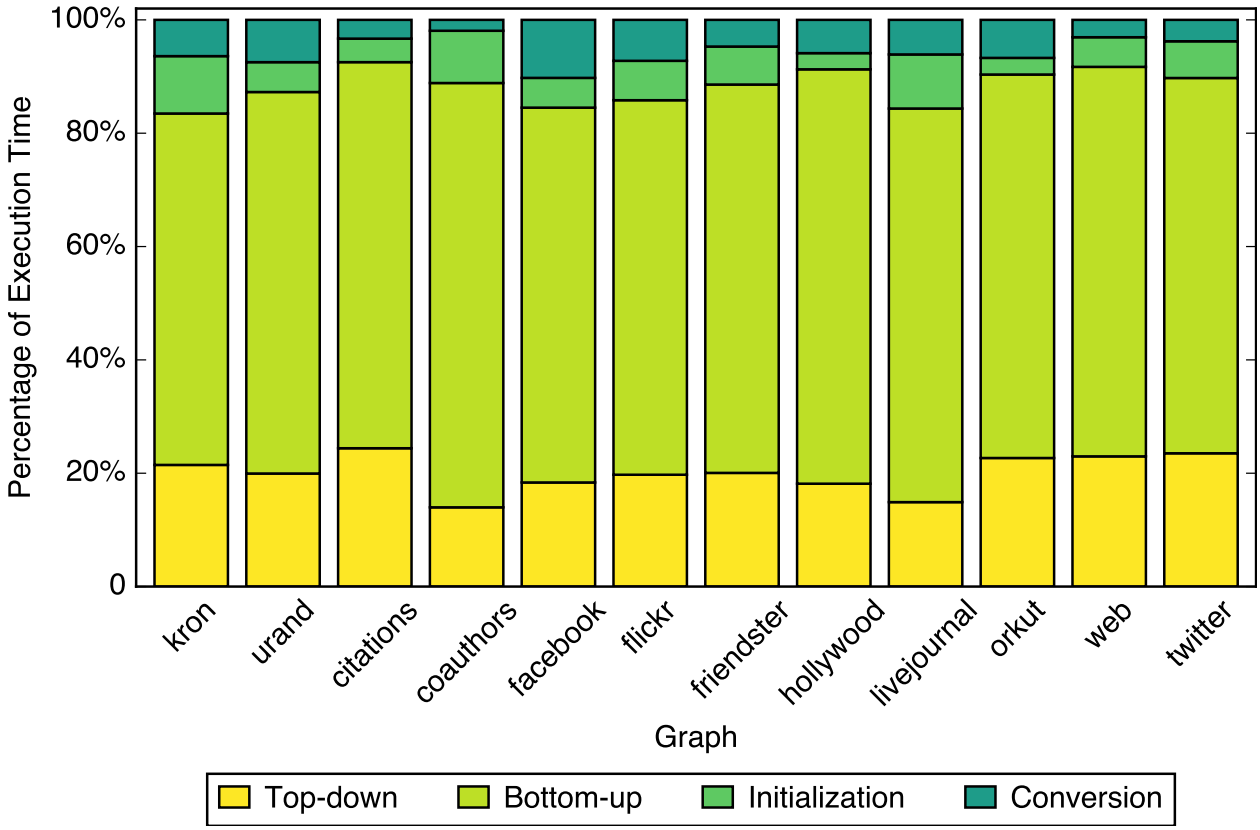

Figure 3.14: Categorization of execution time by hybrid-heuristic

Figure 3.13 plots the types of edges examined or skipped by the direction-optimizing implementation relative to the number of edges examined by the top-down baseline for all of our input graphs. The edge examinations skipped are split between those skipped in the transition and those skipped by the bottom-up approach. Since the bottom-up approach is used when the frontier is large, the top-down approach in the hybrid implementation processes only a small fraction of the edges. Unsurprisingly, the graphs with the least speedup (citations, coauthors, and web) skip fewer edges and have the most wasteful examinations. These graphs have a substantially higher effective diameter, causing more steps after the transition (more premature examinations). These graphs also have more vertices and edges outside the main connected component, causing more disconnected examinations.

Examining where the time is spent during an entire search reveals the majority of it is spent in the bottom-up implementation (Figure 3.14). Since the bottom-up approach skips so many examinations, the effective search rate for the bottom-up steps is much higher (order of magnitude) than the top-down approach. Conversion and initialization take a non-negligible fraction of the runtime, but this overhead is worthwhile due to the extreme speedup provided by the bottom-up approach.

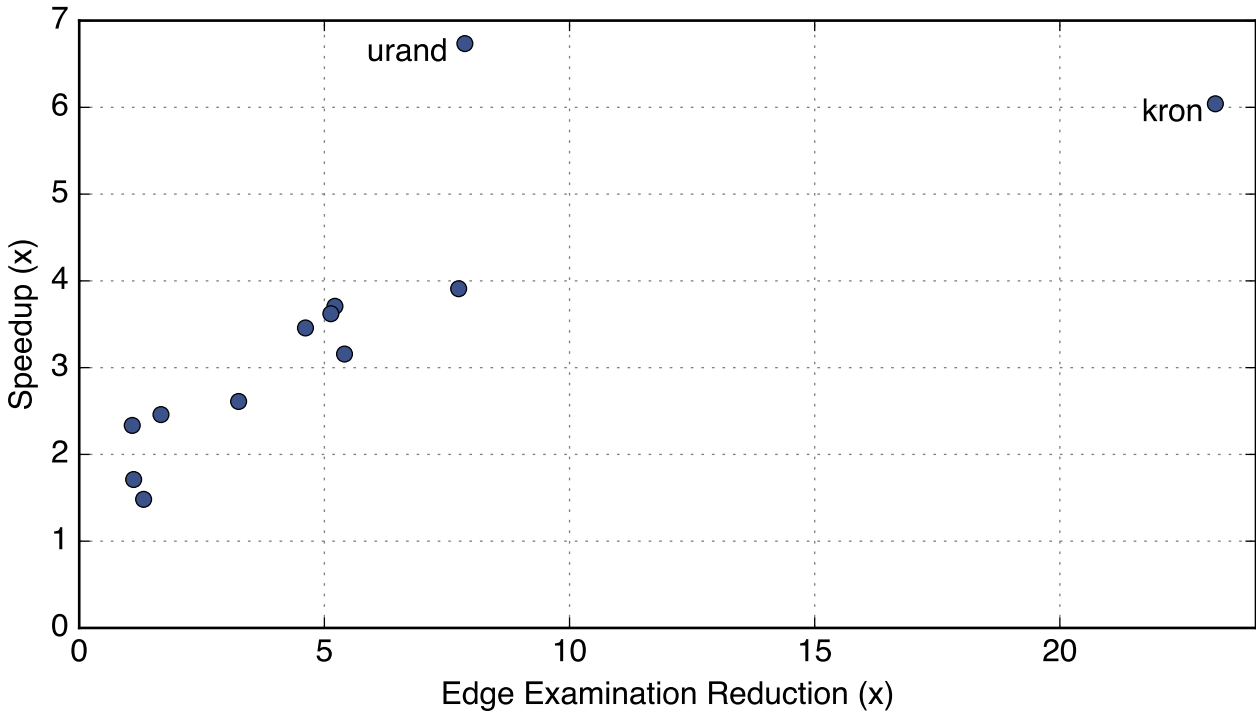

Figure 3.15: Speedups on IVB versus reductions in edges examined for hybrid-heuristic relative to top-down-bitmap baseline.

Figure 3.15 shows the speedups for the suite is reasonably correlated with the reduction in number of edges examined. There are some notable outliers within Figure 3.15, namely the points for the kron graph and the urand graph. The direction-optimizing approach traversing the kron graph experiences a far greater reduction in edges examined due to the strength of its scale-free property, allowing many edges from high-degree vertices to be skipped (Figure 3.13). The urand graph appears to obtain a disproportionate speedup relative to the reduction in edges examined, however, this is actually due to a slower baseline time from top-down-bitmap rather than an exceptionally fast execution time from hybrid-heuristic. The urand graph induces worse cache locality for the processor due to its large size and its lack of the scale-free property (worse temporal locality). The slope of a best-fit line for Figure 3.15 is approximately 0.2. A slope of less than 1.0 implies the direction-optimizing implementation spends more time on each edge it actually traverses than the top-down-bitmap baseline. In later chapters we discuss the impact of the processor architecture on graph algorithm performance, and in particular in Chapter 6 we explain the difference in actual search rates.

3.6 Discussion

After demonstrating the direction-optimizing approach delivers practical speedups and that these speedups are caused by examining substantially fewer edges, we now turn our attention to explaining how so many edges can be skipped. In particular, we discuss the topological properties that enable so many edges to be skipped and how our heuristics are able to correctly predict when traversing in the bottom-up direction will be more efficient.

An active frontier that constitutes a large fraction of the graph is a critical feature for the direction-optimizing approach's success. A large frontier increases the number of edges that can be skipped on the transition to the bottom-up approach as well as enabling the bottom-up approach itself to skip more edges. The right side of Figure 3.6 shows an extreme example in which the frontier reaches a maximum size of 68% of the graph. The frontier size generally follows this convex shape of increasing in size until an apex and then decreasing in size. Arbitrary synthetic graphs can break this trend, but in general, real-world graphs have this shape due to statistical averaging of distances. The average degree of the frontier is not constant across all depths, so the frontier is largest in terms of its constituent edges or its constituent vertices on different depths. In particular, the frontier size often reaches its apex in terms of edges one depth less than the apex in terms of vertices. This relationship is also typical, as the high-degree vertices are more interconnected and thus reached sooner in the traversal. Once the high-degree vertices are traversed, the search continues to their many low-degree neighbors, which results in the vertex frontier apex one depth later.

For the bottom-up approach to reduce the number of edges examined, it must skip more edges than it examines redundantly. More specifically, the number of edges skipped during the transition from top-down to bottom-up plus the number of edges skipped in the first bottom-up step must be less than the overhead of examining necessary edges (premature or disconnected). In general, the best depth to switch to bottom-up is when the frontier is at its largest in terms of edges. This maximizes the number of edges skipped in the transition, as well as providing a large edge frontier to increase the probability of skipping edges during the first bottom-up step.

It is challenging to predict if the current step is the edge frontier apex and thus the appropriate time to switch to the bottom-up approach because the size of the next edge frontier is not yet known. Our practical heuristic decides the frontier is sufficiently large (and probably the apex) if the number of edges exceeds $1/\alpha^{th}$ of the remaining edges. The size of the edge frontier apex and the depth at which it occurs varies and depends not only on the input graph, but also the source vertex for the BFS traversal. By picking a single α for a hardware platform and using it for all graphs, we are attempting to minimize the number of mispredictions by achieving the right balance between false positives (switching early) and false negatives (switching late). Switching early (α too big) not only reduces the number of edges skipped on transition, but the first bottom-up step will be slower as a smaller frontier reduces the probability it can skip edges. Switching late (α too small) causes the last top-down step to process the many edges that would have been skipped altogether on the transition. It is also important to make sure the bottom-up approach is not used at

all if the frontier never gets large enough, which typically happens on high-diameter graphs. Overall, our pragmatic heuristic along with tuning with the implementations used on the target hardware does well at this, as exemplified by the wide range of acceptable values for α in Figure 3.9.

Once the direction-optimizing approach has switched to using the bottom-up approach, it is usually profitable to stay in the bottom-up approach for a few additional steps. After the first bottom-up step, the savings in edge examinations from the transition have already accrued and thus whether it is profitable to stay with the bottom-up approach for an additional step depends solely on whether the bottom-up approach for that step will examine fewer edges than the top-down approach for that step. The bottom-up approach will examine fewer edges if the number of edges it skips by early termination are greater than the number of examinations of premature and disconnected edges. This comparison is typically advantageous for the bottom-up approach if the frontier size is rapidly shrinking, as the future redundant work is outnumbered by the current savings. During a traversal, it is often hard to predict how big the next depth will be, so our other practical heuristic decides the frontier is too small for the bottom-up approach if it contains fewer than $1/\beta^{th}$ of the graph's vertices. We consider the number of vertices instead of the number of edges because our bottom-up implementation scans all of the vertices to find unvisited vertices, so that fixed cost for scanning is only worthwhile if the frontier is sufficiently large.

We increase the robustness of our control heuristic by sticking with the bottom-up approach if the number of vertices in the frontier is growing. If the traversal happens to prematurely switch to the bottom-up approach, the worst thing that could happen is if the traversal also prematurely switches back to top-down before switching to the bottom-up approach a second time. Transitioning to bottom-up twice will examine more edges than if the traversal stubbornly remains in the bottom-up state. The second bottom-up switch will only be motivated if the frontier continues to grow, and so using frontier growth as an additional condition is able to avoid this perfromance pathology. This condition will of course not affect the typical transition from bottom-up to top-down when the frontier is shrinking, and it only has an impact on a few input graphs (e.g. coauthors). High-diameter graphs have more gradual frontier growth rates and frontier shrinking rates, and are thus the most susceptible to this premature transition that could lead to a costly double-transition, but the growing condition is able to avoid most cases.

Fundamentally, the most important topological property for the success of the direction-optimizing approach is the apex of the frontier size. The larger the largest frontier size, the more edges the direction-optimizing approach can skip. Such sharp frontier apexes occur most often in low-diameter graphs since such a large fraction of the graph is reached in so few steps. The success of the direction-optimizing approach (Figure 3.8) is indicative of how common these sharp frontier apexes are in real-world low-diameter graphs.

3.7 Related Work

Buluç and Madduri [33] provide a thorough taxonomy of related work in parallel breadth-first searches. In this section, we focus on the work most relevant to this study, principally parallel shared-memory implementations. Much of the work prior to the direction-optimizing approach focuses on improving performance by increasing parallelism or locality.

Bader and Madduri [9] demonstrate large parallel speedups for BFS on an MTA-2. Parallelism is extracted both at the level of each vertex as well as at each edge check, and the fine-grained synchronization mechanisms of the MTA are used to efficiently exploit this parallelism. Their implementation does not need to optimize for locality since the MTA does not use caches and instead uses many hardware threads to hide its shared main memory latency.

In contrast, Agarwal et al. [2] optimize for locality to push the limits of a quad-socket system built from conventional microprocessors, and show speedups over previous results from clusters and custom supercomputers, including the MTA-2 [9]. Careful programming is used to reduce off-socket traffic (memory and inter-socket) as much as possible. Threads are pinned to sockets and utilize local data structures such that all the work that can be done within a socket never leaves the socket. Bitmaps are used to track completed vertices to avoid accessing DRAM. Whenever an edge check must go to another socket, it utilizes a custom inter-socket messaging queue.

Hong et al. [77] improve upon Agarwal et al. with a hybrid algorithm that utilize multiple CPU implementations and a GPU implementation. As in our work, the algorithm switches at step boundaries and makes decisions based on on-line heuristics, but in this case to select between CPU and GPU implementations of a purely top-down approach. The GPU implementation outperforms the CPU implementation for a sufficiently large frontier, and heuristics select which implementation to use. The CPU implementation is accelerated by the read-array approach, which improves locality by combining the frontier with the output array. Their implementation outputs the depth of each vertex (instead of the parent like ours), so the output depth array can also act as the frontier. On each step, the depths of all vertices are scanned searching for the current depth, and then edge examinations are performed from any vertex found in the implicit frontier. The action of setting an unvisited vertex's output to depth+1 implicitly adds it to the next frontier. Furthermore, since Hong et al. bound the depth to be less than 256, the output array can be compacted by using only one byte per vertex. Best of all, by using the output array to act as the frontier, duplicate entries are squashed, and spatial locality is increased due to the sequential array accesses to the graph.

Merrill et al. [106] improve the performance of BFS on GPUs through the use of prefix sum to achieve high utilization of all threads. The prefix sum is used to compute the offsets from the frontier expansion, which reduces contention for atomic updates to the frontier queue. Using a prefix sum to allocate work to threads also allows the graph exploration to have little control divergence between threads. They also leverage various filtering techniques enabled by bitmaps to reduce the number of edges processed. This approach is beneficial for

all diameters of graphs, and they present results indicating they are the fastest at the time of publication for shared memory systems, especially with their quad-GPU parallelization.

Chhugani et al. [40] perform a multitude of optimizations to improve memory utilization, reduce inter-socket traffic, and balance work between sockets. Their memory performance is improved by reordering computation and data layout to greatly increase spatial locality. The load balancing techniques proposed are dynamic and can adapt to the needs of the current graph. Furthermore, they demonstrate an analytic model derived from the platform's architectural parameters that accurately predicts performance. Many of these optimizations are complementary to our work and could be added on top of our implementation.

An early implementation of our direction-optimizing algorithm on a single-node system reached 17^{th} place in the November 2011 rankings of the Graph500 competition [15, 66]. This earlier implementation used a cruder heuristic and included an alternate top-down step that integrated the conversion for bottom-up. It achieved the highest single-node performance and the highest per-core processing rate. Using just a single quad-socket system, the hybrid BFS algorithm outperformed clusters of >150 sockets and specialized architectures such as the Cray XMT2 [111] and the Convey HC-1^{ex} [45]. Its processing rate per-core was over 30× the top-ranked cluster system, highlighting the performance penalty a cluster experiences when crossing the network.

This chapter presents the current state-of-the-art of direction-optimizing breadth-first search, and it contains simplifications and refinements to the algorithm since it was first formally published [16]. As an improvement, we reduce the overhead of computing the number of outgoing edges of the frontier (n_f) by using the optimization described in Section 3.5. Furthermore, the results in this chapter also include larger real-world graphs that became available since the original publication. For consistency with the rest of this dissertation, these results have also been executed on IVB (newer hardware). Finally, the implementation used in this chapter is available as part of the GAP Benchmark Suite [60]. The original publication provides direct performance comparisons to the aforementioned prior work (Hong et al. [77], Merrill et al. [106] and Chhugani et al. [40]) demonstrating the direction-optimizing BFS's performance advantage when using the same input graphs and similar if not identical hardware [16].

We also extend our direction-optimizing BFS algorithm to make it appropriate for clusters [20, 32]. The main contribution of the cluster research is a distributed bottom-up implementation, since it can be combined with an existing distributed top-down implementation to obtain a distributed direction-optimizing BFS implementation. Implementing the bottom-up search strategy on distributed memory poses multiple challenges. First, the bottom-up approach needs fast frontier membership tests to find a neighbor in the frontier, but the frontier is far too large to replicate in each processor's memory. Second, each vertex's search for a parent must be sequentialized in order to skip checking unnecessary edges once a parent is found. If a vertex's search for a parent is fully parallelized, there is potential the search will not terminate as soon as a parent is found, resulting in redundant work that could nullify any performance gains. We tackle the first challenge by adapting the two-dimensional graph partitioning approach that reduces the amount of the frontier that needs to be locally

replicated for each processor [33, 173]. We tackle the second challenge by using systolic shifts that provide a good compromise between work efficiency and parallelism. Overall, we obtain speedups over a purely top-down implementation similar to what we obtain in this chapter on shared memory. By reducing the number of edges examined, our implementation not only reduces the amount of computation, but it also reduces the amount of communication which is critical for distributed memory. Due to the sub-linear performance scaling that is typical for distributed graph computation, by using our distributed BFS implementation, one can use substantially fewer processors to achieve the same performance. This implementation is now available as part of the CombBLAS framework [31].

The direction-optimizing approach has been widely adopted, as a number of noteworthy graph frameworks for different hardware platforms contain implementations, including Galois (shared memory) [122], Gunrock (GPU) [158], and Grappa (cluster) [118]. The direction-optimizing approach has also become widely adopted in the Graph500 competition [66], including the IBM implementations [38] and CREST implementations [59] that frequently dominate the top of the rankings. Additionally, an implementation of the direction-optimizing algorithm is now the reference code for the Graph500 competition [66]. The Ligra framework generalizes the direction-optimizing concept to other graph algorithms [146]. The Ligra framework takes care of the decision of whether to go top-down (push) or bottom-up (pull) and it also takes care of the conversion between appropriate frontier data structures. Like this work, Ligra is only advantageous for low-diameter graphs but delivers equivalent performance for high-diameter graphs.

3.8 Conclusion

A low-diameter graph features short paths between nearly every pair of vertices. The multitude of these short paths lead to great interconnectedness amongst the vertices, making many of these paths redundant. This path redundancy manifests itself during a top-down traversal when the frontier is large and most edge examinations fail to discover an unvisited vertex. Most of the edges within this large frontier stay within the frontier (peer edges) and are thus unnecessary to examine since they do not help identify the next frontier. Transitioning to the bottom-up approach avoids the redundancy of the large frontier by considering the remainder of the graph which is less redundant. With the help of the bottom-up approach, the direction-optimizing approach is able to dramatically reduce the number of edges examined to deliver a substantial speedup.

The results of this work demonstrate the performance improvement potential of integrating the bottom-up approach into BFS, but it will not always be advantageous. Fortunately, the same top-down implementation that is used for the steps when the frontier is small can be used for entire searches when the bottom-up approach is not advantageous. The heuristic will allow this to happen automatically, as searches that do not generate a massive frontier will not trigger the transition. In this sense, the bottom-up approach can be seen as a low-risk, powerful way to accelerate a BFS on a low-diameter graph.

High-diameter graphs will not benefit from the bottom-up approach, but they are much easier to partition and rearrange for locality, making them easier to parallelize than low-diameter graphs. This work presents an algorithmic innovation to accelerate the processing of the more difficult-to-parallelize graphs for BFS.

Our direction-optimizing approach yields the current best-known BFS performance for low-diameter graphs. The success of transitioning to the bottom-up approach on a large frontier can be interpreted as a philosophical lesson: When confronted with a great deal of work to do, instead consider the desired result and work backwards.

Chapter 4

GAP Benchmark Suite

In this chapter, we present our Graph Algorithm Platform (GAP) Benchmark Suite to address the shortcomings we observe in graph processing research evaluations. In addition to specifying the benchmark and describing the reference implementation, we also include the rationale for many of our decisions and recount the lessons learned from prior work. In Chapter 5, we use the workload specified by this benchmark for our characterization study. Our benchmark specification and reference implementation are available online (`gap.cs.berkeley.edu/benchmark.html`).

4.1 Introduction

A lack of a standard for evaluations has clouded the results from the growing body of graph processing research. To compare a new result to prior work, ideally everything other than the new contribution should be the same. Unfortunately, there is often insufficient overlap between different published results to make such meaningful comparisons. Simple methodology differences (e.g., treating input edges as directed or undirected) can impact performance by more than the claimed improvement. Even for a particular well-known graph problem, there are often many variations (e.g., tracking parent vertices or vertex depths for breadth-first search) that alter what optimizations are possible. The input graphs themselves can be misleading, as similar or identical names can refer to very different graph datasets. A standard for graph processing evaluations could combat these problems.

Other graph processing evaluation mistakes would be harder to make if there is a well-known evaluation standard. With a standard set of diverse input graphs, if an optimization is only compatible with certain topologies, this weakness could be exposed. A standard set of input graphs could also prevent the use of graphs that are trivially small or unrealistically synthetically generated. A standard high-quality reference implementation could help discourage the use of low-performance baselines.

We present the GAP Benchmark Suite to ameliorate these evaluation concerns. The benchmark specifies graph kernels, input graphs, and measurement methodologies. The

benchmark also includes an optimized reference implementation that is representative of state-of-the-art performance [60]. To create our benchmark, we learned from the best practices of the community, and we co-developed this benchmark with our graph workload characterization (Chapter 5). An important lesson from our characterization is the importance of putting together a diverse workload consisting of multiple kernels and input graphs.

A key aspect of our benchmark is the benchmark specification [18]. Other research efforts have released their code [105, 122, 146, 153], which eases comparisons to themselves, but the evaluator is still responsible for creating a workload and an evaluation methodology. Furthermore, since these frameworks were developed independently, they may require some modification to ensure the implementations are computing the same thing and using the same timing practices.

The benchmark specification and the reference implementation are two separate artifacts that can be used independently. By specifying the benchmark explicitly, any implementation on any platform that conforms to the benchmark's specifications can be compared. Consequently, these benchmark-compliant comparisons do not require the use of our reference implementation. Additionally, the reference implementation can be used to execute workloads other than the benchmark.

This benchmark suite can be used in a variety of settings. Graph framework developers can demonstrate the generality of their programming model by implementing all of the benchmark's kernels and delivering competitive performance on all of the benchmark's graphs. Algorithm designers can use the input graphs as a workload and the reference implementation as a baseline to demonstrate their contribution. Platform designers and performance analysts can use the suite as a workload representative of graph processing.

Our reference implementation targets shared memory multiprocessor systems, but that platform is not required to use this benchmark. Our benchmark focuses on the execution of the graph kernels and does not consider the time required to load the graph data or to build the graph itself. Different platforms might load the graphs in different manners, but the input graphs, kernels, and methodologies are all platform-agnostic. Semi-external memory approaches could demonstrate their competitiveness with shared memory baselines. A distributed graph framework should prove itself worthy of cluster resources by substantially outperforming an optimized baseline running on a single node instead of only comparing against itself running on a single node [105].

4.2 Related Work

There have been prior attempts at graph benchmark suites, and we try to leverage their strengths. Unfortunately, none of the prior benchmarks are without weaknesses, and we created our GAP Benchmark Suite to improve graph processing evaluations.

The Graph 500 [66] competition has been a great success for the graph community, so we use its strengths and expand upon them. Graph 500 has strong community adoption that has led to innovation in both algorithms and implementations. For example, the top

finisher from the first competition in November 2010 is 5450× slower than the most recent top finisher (June 2016), and it would place no higher than 116^{th} in the current rankings. The biggest shortcomings of Graph 500 are its focus on one kernel (breadth-first search) and only one synthetic input graph topology (Kronecker). Although there are efforts to add additional kernels to Graph 500, they are still under review [67].

The Scalable Synthetic Compact Applications #2 (SSCA) [12] specifies a synthetic application consisting of four graph kernels. These kernels combined are intended to model a graph analysis workload, and they operate on a single synthetically-generated RMAT [36] graph. The four kernels are: building the graph, finding the largest weight edges, extracting subgraphs, and computing betweenness centrality scores. SSCA is a well-designed benchmark, which includes both a specification and clear reference code. We improve upon SSCA by considering more graph kernels and more input graph topologies. In particular, our input graphs include real-world datasets.

The Problem-based Benchmark Suite (PBBS) [147] is another noteworthy benchmark effort. We improve upon it by providing substantially higher performance reference code, using real-world graphs, and focussing on graph algorithms alone.

After our benchmark's initial release, two more graph benchmarks have been released: CRONO [3] and GraphBIG [116]. CRONO is designed for future tiled manycore systems, while GraphBIG supports both conventional multiprocessors and GPUs. GraphBIG also provides a useful taxonomy of graph kernels, and GraphBIG's greatest strength is its focus on applications. Unfortunately, both benchmarks make some of the mistakes our benchmark is designed to fix. Both CRONO and GraphBIG do not explicitly specify what they benchmark, which complicates comparisons because users must examine their code to infer what they are benchmarking. Additionally, the input graphs used by both benchmarks are small relative to the platforms they execute on. The largest graphs in our benchmark are 50× larger than the largest graph in CRONO and 25× larger than the largest graph in GraphBIG. Finally, their included reference implementations do not provide competitive performance.

4.3 Benchmark Specification

This work is motivated by the evaluation shortcomings we observe in prior work. We created this benchmark based on our experiences competing in Graph500 [15], performing our workload characterization (Chapter 5), and analyzing prior work (Appendix A). We designed this benchmark with the following goals in mind, each intended to solve existing evaluation problems:

- Explicit benchmark specifications to standardize evaluations in order to improve comparability and reduce confusion
- Diverse representative workload to ensure evaluations have relevant target

- High-quality reference implementation to ground evaluations with strong baseline performance

In this section, we specify the benchmark by describing the graph kernels and input graphs. We also describe the required evaluation methodologies and provide rationale for our decisions.

Graph Kernels

We select six kernels based on how commonly they are used (Appendix A). These kernels are representative of many applications within social network analysis, engineering, and science. This set of kernels is computationally diverse, as it includes both traversal-centric and compute-centric kernels. Across the suite, different kernels will consider or ignore different properties of the graph including edge weights and edge directions. For more background on what the algorithms compute, please consult Section 2.4. To remove ambiguity, we describe which variant of each graph problem we require and what constitutes a correct solution.

- **Breadth-First Search (BFS)**
  We make BFS into a kernel by tracking the parent vertices during the traversal, akin to Graph 500 [66]. For any reached vertex, there is often more than one possible parent vertex, as any incoming neighbor with a depth one less than the reached vertex could be a parent. Multiple legal parent vertices cause there to be more than one correct solution to BFS from a given source vertex. For this reason, we define the correct solution to BFS starting from a *source* vertex to be a *parent* array that satisfies the following:
  - parent[source] = source
  - parent[v] = -1 if v is unreachable from source
  - if v is reachable and parent[v] = u, there exists an edge from u to v
  - if v is reachable and parent[v] = u, depth[v] = depth[u] + 1

  Some uses of BFS only track reachability or depth, but we choose to track the parent since it is the most helpful to the community. If we instead chose to track reachability, there would be no way to verify the traversal was performed in a breadth-first manner, because reachability only returns a boolean value for each vertex. Tracking depth would be a better choice than reachability since it allows for verifying the traversal, but BFS implementations that track depth can perform optimizations not possible in the more challenging BFS implementations that track parents [106].

- **Single-Source Shortest Paths (SSSP)**
  For SSSP, we require the distances of the shortest paths. We do not request the parent vertices, since BFS already provides this. There is a unique solution to our variant

of SSSP, since the solution is the distances and not the shortest paths themselves. Although there may be more than one shortest path between two vertices, all shortest paths will have the same distance. All of our benchmark graphs have positive edge weights. We define the correct solution of SSSP to be the *distance* array from a *source* vertex such that:

  - distance[source] = 0
  - distance[v] = ∞ (or some known sentinel value) if v is unreachable from source
  - if v is reachable from source, there is no path of combined weight less than distance[v] from the source to v

- **PageRank (PR)**
  For PR, we require the PageRank scores for all of the vertices such that a single additional iteration will change all of the scores by a sum of less than 10^{-4}. More formally, if the benchmark kernel returns scores PR_k and a classical implementation generates PR_{k+1} in one iteration from PR_k, a solution is correct if it obeys the tolerance:

$$\sum_{v \in V} |PR_k(v) - PR_{k+1}(v)| < 10^{-4}$$

  Selecting the required tolerance is a tradeoff between score convergence and execution time. We pragmatically select 10^{-4} since the scores will have mostly converged and it results in a reasonable number of iterations (5 - 20) for most graphs. Our tolerance bound also implicitly allows for a little numerical noise due to differences in accumulation order. We allow for more advanced implementations as long as they meet this bound and any changes to the graph or preprocessing optimizations are included in the trial time.

- **Connected Components (CC)**
  For CC, we require all vertices to be labelled by their connected component and each connected component to get its own unique label. If the graph is directed, we only require weakly connected components, so if two vertices are in the same connected component, it is equivalent to there being a path between the two vertices if the graph's edges are interpreted as undirected. Vertices of zero degree (disconnected) should each get their own label. To define correctness, we require the following equivalence relation:

  - vertices u and v have the same component label if and only if there exists an undirected path between u and v

- **Betweenness Centrality (BC)**
  For BC, we require the approximate BC scores for all vertices, and these centrality scores should be normalized to one. For the approximation, the BC scores should be computed by considering the shortest paths from 4 different sources. We expect

most implementations to accomplish this by using Brandes algorithm [29] and performing 4 BFS traversals, but the use of Brandes algorithm is not required. The BC implementation should treat all input graphs as unweighted.

For correctness, we recommend comparing the output to the output from a simple correct implementation of an alternate algorithm using the same source vertices. In this case, the verifier may need to allow for a little numerical noise since the accumulations can happen in different orders.

- **Triangle Counting (TC)**
  For TC, we define the correct solution to be the total number of triangles within the input graph. We define a triangle to be three vertices that are directly connected to each other (clique of size 3). A triangle is invariant to permutation, so the same three vertices should be counted as only one triangle no matter the order in which they are listed. Additionally, for our definition of a triangle, we ignore the directions of the edges, so the input graph can be interpreted as undirected. For TC, the solution is unique, so comparing the result is trivial. Unfortunately, there is no easy way to verify the total number of triangles without actually computing it. For verification, we recommend comparing the result from the benchmark implementation with the result from an alternate implementation.

Input Graphs

We select five input graphs for our benchmark and they are diverse in both topology and origin (synthetic versus real-world). Our real-world data models the connections between people, websites, and roads. The graph sizes are selected to be small enough to fit comfortably in most servers' memory yet large enough to be orders of magnitude bigger than the processors' caches. When selecting real-world benchmark graphs, we considered the ease of users acquiring the graph data and the graphs selected are amongst the easiest to obtain both in terms of licensing requirements and bandwidth availability. For more information on the graphs and their topological properties, please consult Section 2.3.

- **Twitter** ($|V|$=61.6M, $|E|$=1,468.4M, directed) is an example of a social network topology [91]. This particular crawl of Twitter has been commonly used by researchers and thus eases comparisons with prior work. By virtue of it coming from real-world data, it has interesting irregularities and the skew in its degree distribution can be a challenge for some implementations.

- **Web** ($|V|$=50.6M, $|E|$=1,949.4M, directed) is a web-crawl of the .sk domain (`sk-2005`) [47]. Despite its large size, it exhibits substantial locality due to its topology and high average degree.

- **Road** ($|V|$=23.9M, $|E|$=58.3M, directed) is all of the roads in the USA [50]. Although it is substantially smaller than the rest of the graphs, it has a high diameter which can cause some synchronous implementations to have long runtimes.
- **Kron** ($|V|$=134.2M, $|E|$=2,111.6M, undirected) uses the Kronecker synthetic graph generator [94] with the same parameters as Graph 500 (A=0.57, B=C=0.19, D=0.05) [66]. It has been used frequently in research due to Graph 500, so it also provides continuity with prior work.
- **Urand** ($|V|$=134.2M, $|E|$=2,147.4M, undirected) is synthetically generated by the Erdős–Réyni model (Uniform Random) [58]. With respect to locality, it represents the worst case as every vertex has equal probability of being a neighbor of every other vertex. When contrasted with the similarly sized kron graph, it demonstrates the impact of kron's scale-free property.

All of the graphs except road are unweighted, so weights must be added to the graphs before executing SSSP. To generate weights for the other graphs, we adopt the practice from Graph 500's SSSP proposal of using uniformly distributed integers from 1 to 255 [67]. We recommend using our reference code for generating these weights as it is deterministic.

Measurement Methodologies

Given the input graphs and kernels, we now specify the measurement methodologies. Building on the success of Graph 500 [66], we reuse many of its best practices.

Executing only a subset of the benchmark kernels is allowed, as some users of this suite may only be investigating a single graph problem. However, it is highly recommended to always use all of the input graphs as they are selected to be diverse. A new innovation may not work well for all input graphs, so it is important to understand the topologies for which the innovation is advantageous.

Since the benchmark does not focus on graph loading or graph building, each trial can assume the graph is already loaded. For example, on a shared-memory multiprocessor, a loaded graph might reside completely in memory in the CSR format. There are no restrictions on the layout of the graph in memory, however, the same graph layout must be used for all kernels. Any optimizations done to the graph layout must not be beneficial to only one algorithm. It is legal to remove duplicate edges and self-loops from the graph. It is also legal to reorder the neighbors of a vertex. If the graph is anyhow transformed or converted from the format used for the other kernels, the graph conversion time should be included in the trial time. If the graph is relabelled, outputs of the graph kernel must use the original vertex labels and this label conversion time should also be included in the trial time.

We select the number of trials each kernel should run with the goal of minimizing evaluation time while capturing enough samples to return significant results. In general, the kernels can be grouped into two classes: "single-source" kernels take a source vertex to start from and "whole-graph" kernels process the entire graph every time in the same way. For

Kernel	Trials	Output per Trial
BFS	64 trials from 64 sources	$\|V\|$-sized array of 32-bit integers (vertex identifiers)
SSSP	64 trials from 64 sources	$\|V\|$-sized array of 32-bit integers (distances)
PR	16 trials	$\|V\|$-sized array of 32-bit floating point numbers
CC	16 trials	$\|V\|$-sized array of 32-bit integers (component labels)
BC	16 trials each from 4 sources	$\|V\|$-sized array of 32-bit floating point numbers
TC	3 trials	64-bit integer (number of triangles)

Table 4.1: Trial counts and output formats for benchmark kernels

single-source kernels (BFS, SSSP, and BC), there is naturally substantial variation in execution time so we execute 64 trials from different source vertices. The source vertices are randomly selected non-zero degree vertices from the graph, and we recommend the vertex selector from the reference code as it is deterministic. For the whole-graph kernels (PR, CC, and TC), we execute just enough trials to capture any performance non-determinism. We reduce the number of trials for TC since most implementations typically have little variance in execution time and the execution time for TC is typically orders of magnitude longer than the rest of the suite. The trial counts are summarized in Table 4.1.

Each trial of a kernel should be timed individually and it should include every aspect of its execution. Each trial can assume the graph is already loaded. Any time to construct data structures other than the graph used by the kernel, including memory allocated for the solution, must be included in the trial time. Additionally, the graph is the only data structure that can be reused between trials, as the purpose of repeated trials is to measure variance, not to amortize optimizations.

For each kernel, there must be only one implementation used for all input graphs. If different approaches will be better for different graph topologies, they should be combined into a hybrid implementation that includes a runtime heuristic (included in kernel time) to decide which approach to use. The same restriction applies to tuning parameters. None of the kernels may take parameters specific to the input graph with the exception of a Δ parameter for SSSP. We allow Δ because it is difficult to achieve high-performance on SSSP without it [128]. Fortunately, when SSSP is used in practice, Δ is available since its determining factors (graph diameter and edge weight distribution) are known within an application domain. For more information on delta-stepping and the impact of Δ, please see Section 6.3. A kernel can take tuning parameters specific to the hardware platform that are the same for all input graphs.

The particular output formats are summarized in Table 4.1. Our benchmark allows for the use of 32-bit vertex identifiers, which is in contrast to Graph 500 which requires at least 48-bit vertex identifiers. Unlike Graph500, for our benchmark, the graphs are of a known size and 32 bits comfortably accommodates them. Using larger identifiers unnecessarily penalizes the performance of cache-based systems, as none of the graphs will utilize more than 28 bits

per identifier. However, benchmark implementations must support graphs with more than 2^{32} edges, and this is typically accomplished with 64-bit pointers. This mix of 32-bit and 64-bit types may seem inconsistent, but this is common practice due to the size of graphs that can fit within the memory of today's multiprocessor platforms. When the memory capacity of systems increases substantially, the input graphs will need to be expanded and this requirement may be increased.

4.4 Reference Implementation

In addition to the benchmark specification, we also provide a reference implementation that includes all six benchmark kernels. This implementation serves multiple purposes. At a minimum, the implementation is compliant with the benchmark specifications and can serve as a high-performance baseline for evaluations. The implementation implements state-of-the-art algorithms that provide it with competitive performance that is the best for some of the kernels. Additionally, the implementation serves an educational purpose, as it clearly demonstrates how to implement some leading algorithms that may not be clearly described by their original algorithmic descriptions. Furthermore, the implementation can help researchers, as much of the infrastructure can be reused as a starting point for high-performance graph algorithm implementations.

To serve all of these purposes, extra effort has been taken to improve code quality. Doing so not only helps those that read it, but it also makes it easier to modify and more portable. Our code follows Google's C++ style guide [65] and uses many best practices for C++ software engineering [82, 108]. Our code leverages many of the features of C++11 that allow us to program safely without any loss of performance. For example, the kernel implementations and most of the core infrastructure do not perform manual memory management or even use pointers. For parallelism, we leverage OpenMP (version 2.5 or later) and restrict ourselves to its simpler features in order to keep our code portable. We have successfully built and run our code on the x86, ARM, SPARC, and RISC-V ISA's using the gcc, clang, icc, and suncc compilers.

The core infrastructure for our reference code includes support for loading graphs from files, synthetically generating graphs, and building graphs in memory. We support multiple file formats, including the popular METIS [86] and Matrix Market formats [25]. As a fallback method for importing graphs into our infrastructure, we also support an extremely simple plain text format that is easy to convert to. Once our infrastructure has built a graph, it can serialize it to a file so later invocations can simply load the serialized graph directly into memory to skip building the graph. Loading the serialized graph saves time and reduces peak memory consumption. We also provide synthetic graph generators for urand and kron. The synthetic graph generators take advantage of seeding and C++11's strict random number generator specifications to deterministically produce the same graph in parallel even on different platforms or with different numbers of threads.

Our reference code includes testing throughout. In addition to testing the code for loading a graph from a file, generating a graph synthetically, or building a graph in memory, our kernel implementations can verify the correctness of their outputs. When there is more than one correct output (BFS, PR, and CC), our verifiers test the output for the properties of a correct solution. For the other kernels (SSSP, BC, and TC), we compare the output from the benchmark implementation to the output of a simple serial implementation that implements the kernel with a different algorithm.

In the remainder of this section we describe some of the most noteworthy or novel optimizations employed by our kernel implementations, but we recommend examining the code itself [60] to answer detailed questions.

- **Breadth-First Search (BFS)**
  For BFS, we implement the state-of-the-art direction-optimizing algorithm [16], and we describe our implementation and its optimizations in great detail in Chapter 3.

  To verify the output of our BFS implementation, we test for the properties this benchmark specifies for BFS. We check that the parent of the source is the source. We check that there is an edge from the parent of v to v. Finally, we use a trivial serial BFS implementation to obtain the depths of all vertices from the source, and we use those depths to check that parents have depth one less than their children.

- **Single-Source Shortest Paths (SSSP)**
  For SSSP, we implement the delta-stepping algorithm [107] with some implementation optimizations from Madduri et al. [100]. For a description of the delta-stepping algorithm and the impact of the Δ parameter, please refer to the third case study in Section 6.3. A common challenge for implementing delta-stepping is implementing the bins used to radix sort the vertices by distance. These bins are challenging to implement correctly because they need to be high-performance and support concurrent insertions. This task is further complicated by determining the sizes of these shared bins. If the bins are allowed to grow, they must be able to grow concurrently. If the bins are sized sufficiently large to not need to grow, there is the possibility of substantial wasted memory capacity.

  We sidestep the challenge of implementing high-performance, concurrent, resizable bins by using thread local bins. Since the bins are thread local, there are no atomicity concerns and we can use existing serial resizable containers. To allow for work-sharing across threads, we use a single shared bin. This shared bin holds the vertices within the current minimum distance range. The shared bin is easy to implement since the threads only need read-only access. Every iteration, we copy all of contents of the thread-local bins for the minimum distance range into the shared bin. Since these aggregations are done in bulk, there is much less contention, which improves performance. Since there is only a single shared bin, it can be over-allocated such that it does not need to grow. Our design decision to use thread-local bins greatly simplifies our implementation as there are no longer parameters to tune for bin size or the number of bins to pre-allocate.

To verify the output of our SSSP implementation, we compare the distances to the output of a simple serial implementation of Dijkstra's algorithm. If the edge weights and distances are floating point, there are fortunately no concerns about numerical reproducibility. All known practical SSSP implementations add the edge weights in the same order, starting from the root vertex continuing along the shortest path.

- **PageRank (PR)**
  For PR, we implement the naive iterative approach that is quite similar to sparse matrix vector multiplication (SpMV). In Chapter 7, we discuss implementation considerations for PR, including the tradeoffs between implementing PR in the push or pull directions. To avoid the use of atomic memory operations, we perform all updates in the pull direction. Unlike the rest of our implementations in this suite, for PR we deliberately choose to not implement the most sophisticated state-of-the-art algorithm in order to be easily comparable. PR is the most commonly used benchmark, and most often it is implemented in this same classical way. By using the classic approach, our implementation can be directly compared to many implementations since they will both perform the same amount of algorithmic work. Furthermore, many optimized implementations not only change the amount of algorithmic work, but in actuality do not obtain the tolerance bounds they advertise. Our benchmark specification allows for optimized PageRank implementations, but they must meet the tolerance bounds. Our implementation computes the tolerance every iteration and can be sure it is met when it decides to terminate.

  We verify the output of our PR implementation by performing an additional iteration and computing the sum of the scores' changes. To contrast from our reference implementation, our verifier's implementation is serial and performs updates in the push direction.

- **Connected Components (CC)**
  For CC, we implement the Shiloach-Vishkin [144] algorithm with parallelization techniques from Bader et al. [11].

  We verify the result of our CC implementation by checking for the equivalence stated in the CC specification. We accomplish this by performing a single traversal for each label. During each traversal, we check that a different label is not encountered. After all of these traversals, we assert that every vertex has been reached by a traversal. If two components share the same label, they will have unreached vertices because there is only a single traversal per label. Vertices of zero degree have their own label so they will each be traversed trivially.

- **Betweenness Centrality (BC)**
  For BC, we implement a fine-grained parallelization of Brandes [29] algorithm with the lock-free improvements from Madduri et al. [101]. To obtain a slight speedup and

a large reduction in memory use, we record the successors identified during the BFS pass in a bitmap instead of a list for each vertex.

We verify the output of our BC implementation by comparing it to the output from a simple serial implementation. Our verifier implementation also uses Brandes algorithm, however, it is implemented in a different way and it does not record the successors during the BFS pass.

- **Triangle Counting (TC)**
  For TC, we implement two well-known optimizations [42]. To count triangles, we sum the sizes of the overlap between a vertex's neighbor list and its neighbors' neighbor lists. Our first optimization leverages our neighbor lists being sorted and terminates these intersection computations once the triangles found will not obey the invariant of $u > v > w$. Terminating these intersection computations early prevents each triangle from being counted six times. Our second optimization relabels the graph by degree, so when the first ordering optimization is applied, we get additional algorithmic savings. Relabelling the graph is compute-intensive, but counting triangles exactly is also compute-intensive so the relabelling optimization is often worthwhile even for a single execution. We find relabelling the graph is typically beneficial for scale-free graphs, and we use a heuristic to decide when to do it. Our heuristic samples the degrees of vertices and decides if the degree distribution is sufficiently skewed by comparing the sample's median degree with the graph's average degree.

  We verify the total from our TC implementation by comparing it the total returned by a trivial serial implementation that uses a standard library implementation of set intersection. Our simple verifier implementation counts each triangle six times so we divide its initial total by six.

4.5 Conclusion

We believe our GAP Benchmark Suite provides a solution to many of the problems plaguing evaluations in the graph processing research community. By separating our benchmark specification from our reference implementation, we allow other compliant implementations to be easily compared. Our benchmark workload is diverse and features large real-world graphs. Finally, our reference implementation provides a high-performance baseline for future research to compare against. We designed our benchmark based on current workloads and platform memory capacities. To allow for future changes necessitated by new workloads or larger systems, we version both the benchmark specification and the reference implementation. We have already updated both more than once. Since our benchmark suite's initial release, the reference implementation has already been used as a baseline to evaluate two new architecture proposals [113, 126]. We hope others use our benchmark to improve their graph processing evaluations.

Chapter 5

Graph Workload Characterization

In this chapter, we characterize the graph processing workload specified by our benchmark in the previous chapter. With our measurements, we determine the most important factors of the hardware platform for graph algorithm performance.

5.1 Introduction

To best understand graph algorithm performance, we analyze the performance of three high-performance graph processing codebases each using a different parallel runtime, and we measure results for these graph libraries using five different graph kernels and a variety of input graphs. We use microbenchmarks and hardware performance counters to analyze the bottlenecks these optimized codes experience when executed on an Intel Ivy Bridge server. We derive the following insights from our analysis:

- *Memory bandwidth is not fully utilized* - Surprisingly, the other bottlenecks described below prevent the off-chip memory system from achieving full utilization on well-tuned parallel graph codes. In other words, there is the potential for significant performance improvement on graph codes with current off-chip memory systems.

- *Graph codes exhibit substantial locality* - Optimized graph codes experience a moderately high last-level cache (LLC) hit rate.

- *Reorder buffer size limits achievable memory throughput* - The relatively high LLC hit rate implies many instructions are executed for each LLC miss. These instructions fill the reorder buffer in the core, preventing future loads that will miss in the LLC from issuing early, resulting in unused memory bandwidth.

- *Multithreading has limited potential for graph processing* - In the context of a large superscalar out-of-order multicore, we see only modest room for performance improvement on graph codes from multithreading, and that is likely achievable with only a modest number of threads (two) per core.

Graph	Description	Vertices (M)	Edges (M)	Degree	Degree Distribution	Diameter
Road	USA road network [50]	23.9	58.3	2.4	bounded	6,277
Twitter	social network [91]	61.6	1,468.4	23.8	power	7
Web	crawl of .sk domain [47]	50.6	1,949.4	38.5	power	13
Kron	Kronecker generator [66, 94]	134.2	2,111.6	15.7	power	5
Urand	uniform random [58]	134.2	2,147.4	16.0	normal	6

Table 5.1: Graphs used for characterization

We also confirm conventional wisdom that the most efficient algorithms are often the hardest to parallelize, and that these algorithms have their scaling hampered by load imbalance, synchronization overheads, and non-uniform memory access (NUMA) penalties. Additionally, we find that different input graph sizes and topologies can lead to very different conclusions for algorithms and architectures, so it is important to consider a range of input graphs in any analysis.

Based on insights from our empirical results, we make recommendations for future work in both hardware and software to improve graph algorithm performance.

5.2 Methodology

To generate a representative graph workload, we select the graph kernels and input graphs from the GAP Benchmark and execute them with three different high-performance graph codebases running on a modern high-end server. Unless otherwise stated, we measure the full workload for each system configuration on all 75 combinations of codebases (3), kernels (5), and input graphs (5).

Characterization Workload: Kernels, Graphs, and Frameworks

The five graph kernels we selected from the GAP Benchmark are:

- Breadth-First Search (BFS)
- Single-Source Shortest Paths (SSSP)
- PageRank (PR)
- Connected Components (CC)
- Betweenness Centrality (BC)

Core		System	
Architecture	Ivy Bridge EP	Released	Q3 2013
Intel Model	E5-2667 v2	# Sockets	2
Clock rate	3.3 GHz	Cores/socket	8
Threads/core	2	LLC/socket	25 MB
TLB 4 KB Entries	64 L1, 512 L2	DRAM Capacity	128 GB
TLB 2 MB Entries	32	DRAM Type	DDR3-1600
TLB 1 GB Entries	4	DRAM DIMMs	16

Table 5.2: Specifications for IVB system used for characterization

We omit Triangle Counting as it is not available in two of the codebases and its runtime is too onerous when executed by only a single core. We use the five input graphs from the GAP Benchmark, and we re-list them in Table 5.1. The GAP suite is diverse, as it includes both real-world and synthetic data, both low-diameter and high-diameter graphs, and both mesh and social network topologies. Please refer to Chapter 2 for more information on the graphs or algorithms and Chapter 4 for the GAP Benchmark's rationale for selecting them.

For this study, we use three of the fastest graph codebases available, which each use a different parallel runtime.

Galois uses its own custom parallel runtime specifically designed to handle irregular fine-grained task parallelism [122]. Algorithms implemented in Galois are free to use autonomous scheduling (no synchronization barriers), which should reduce the synchronization otherwise needed for high-diameter graphs. Additionally, Galois' scheduler takes into consideration the platform's core and socket topology.

Ligra uses the Cilk [24] parallel runtime and is built on the abstractions of edge maps and vertex maps [146]. When applying these map functions, Ligra uses heuristics to determine in which direction to apply them (push or pull) and what data structures to use (sparse or dense). These optimizations make Ligra especially well suited for low-diameter graphs.

GAP Benchmark Suite (GAPBS) is our reference implementation [18, 60] for the GAP Benchmark (Section 4.4). GAPBS is a collection of high-performance implementations written directly in OpenMP with C++11. Since GAPBS is not a framework, it does not force common abstractions onto all implementations, but instead frees each to do whatever is appropriate for a given algorithm.

All three codebases are competitive, and depending on the input graph or kernel, a different codebase is the fastest. For descriptions of the implementations and their parallelization strategies, we refer the reader to the original publications.

Hardware Platform

To perform our measurements, we use a dual-socket Intel Ivy Bridge server (IVB), similar to what one would find in a datacenter (Table 5.2). To access hardware performance counters, we use Intel PCM [81] and PAPI [112]. We compile all code with gcc-4.8, except Ligra, which uses Cilk Plus gcc-4.8. To ensure consistency across runs, we disable Turbo Boost (dynamic voltage and frequency scaling).

When reporting memory traffic from the performance counters, we focus on memory requests caused by LLC misses, as these are the most problematic for performance. We do not include prefetch traffic measurements because they obscure the results, but benefits of successful prefetching appear indirectly as fewer cache misses. During our study, we observed IVB intelligently prefetching aggressively when the memory bandwidth utilization would otherwise be low, but ceasing prefetching when the application is using a large fraction of the memory bandwidth. That is the hardware prefetcher does not prevent full memory bandwidth utilization.

5.3 Memory Bandwidth Potential

Any LLC miss will cause even a large out-of-order processor to stall for a significant number of cycles. Ideally, while waiting for the first cache miss to resolve, at least some useful work could be done, including initiating loads early that will cause future cache misses. Unfortunately, a load must satisfy the following four requirements before it can be issued:

1. *Processor fetches load instruction* - Control flow reaches the load instruction (possibly speculatively).
2. *Space in instruction window* - The Reorder Buffer (ROB) is not full.
3. *Register operands are available* - The load address can be generated.
4. *Memory bandwidth is available* - At the core level there is a miss-status holding register (MSHR) available and there is not excessive contention within the on-chip interconnect or at the memory controller.

If any of the above requirements are not met, the load will be unable to issue. In particular, memory bandwidth cannot be a bottleneck unless the first three requirements are satisfied, thus the other factors can prevent memory bandwidth from being fully utilized.

We execute a parallel pointer-chase as a synthetic microbenchmark on our IVB platform to understand the interactions between the requirements for issuing a load and to quantify the platform's limits. A parallel pointer-chase exposes the needed parameters to control which load requirements are in effect, but is otherwise quite simple [2, 120]. With a single pointer-chase, there is no memory-level parallelism (MLP) and the memory latency is exposed since requests must be completed serially. To generate more MLP, we simply add more parallel pointer chases to the same thread (Figure 5.1).

```
for (int i=0; i<n; i++) {
   ptr1 = *ptr1
   ptr2 = *ptr2
   :
   ptrk = *ptrk
}
```

Figure 5.1: Parallel pointer-chase microbenchmark for k-way MLP

To force loads to access the memory system beyond the on-chip caches, we set pointers to point randomly within an array sized large enough such that LLC hit rates are less than 1.5% (typically $\geq$ 2 GB). We report bandwidths in terms of memory references per second as measured by performance counters. We also report achieved bandwidths in terms of *effective MLP*, which is the average number of memory requests in flight according to Little's Law (memory bandwidth $\times$ memory latency). It is worth distinguishing this from *application MLP*, which is how much memory-request parallelism is allowed by the application's data dependencies, which will be always greater than or equal to the achieved effective MLP.

Our simple microbenchmark is designed to trivially satisfy the first two requirements above, allowing us to focus on and measure the last two. Branch mispredictions should be rare since the loop repeats many times, so fetching the load instructions should not be hindered. The microbenchmark is a tight loop, so there should be a relatively high density of loads and the microbenchmark randomly accesses a large array, so the vast majority of the loads should miss the LLC. Frequent cache-missing loads reduces the impact of the instruction window size (168 for Ivy Bridge). By changing the number of parallel pointer-chases, we can artificially control the maximum application MLP possible, which allows us to moderate the operand availability requirement. We can then observe what bandwidths are possible and even what the bandwidth limits are.

Figure 5.2 shows the microbenchmark results for a single core. The local memory latency is 86 ns (MLP=1). Local bandwidth for a single thread appears to saturate when MLP$\geq$10, implying the core supports 10 outstanding misses, and this is confirmed by published sources on the Ivy Bridge microarchitecture [80]. Using a second thread on the same core does not change the maximum bandwidth regardless of how the outstanding memory requests are spread across the two threads.

To see the impact of Non-Uniform Memory Access (NUMA) on our dual-socket system, instead of allocating the memory being used by our microbenchmark on the same socket (local), we allocate on the other socket (remote) or interleaved across both sockets (interleaved). NUMA may introduce bandwidth restrictions, but for a single core in isolation, the primary consequence is a doubling of the latency ($\approx$184 ns). When accessing remote memory, the maximum bandwidth is halved due to the same number of outstanding data requests experiencing twice the latency.

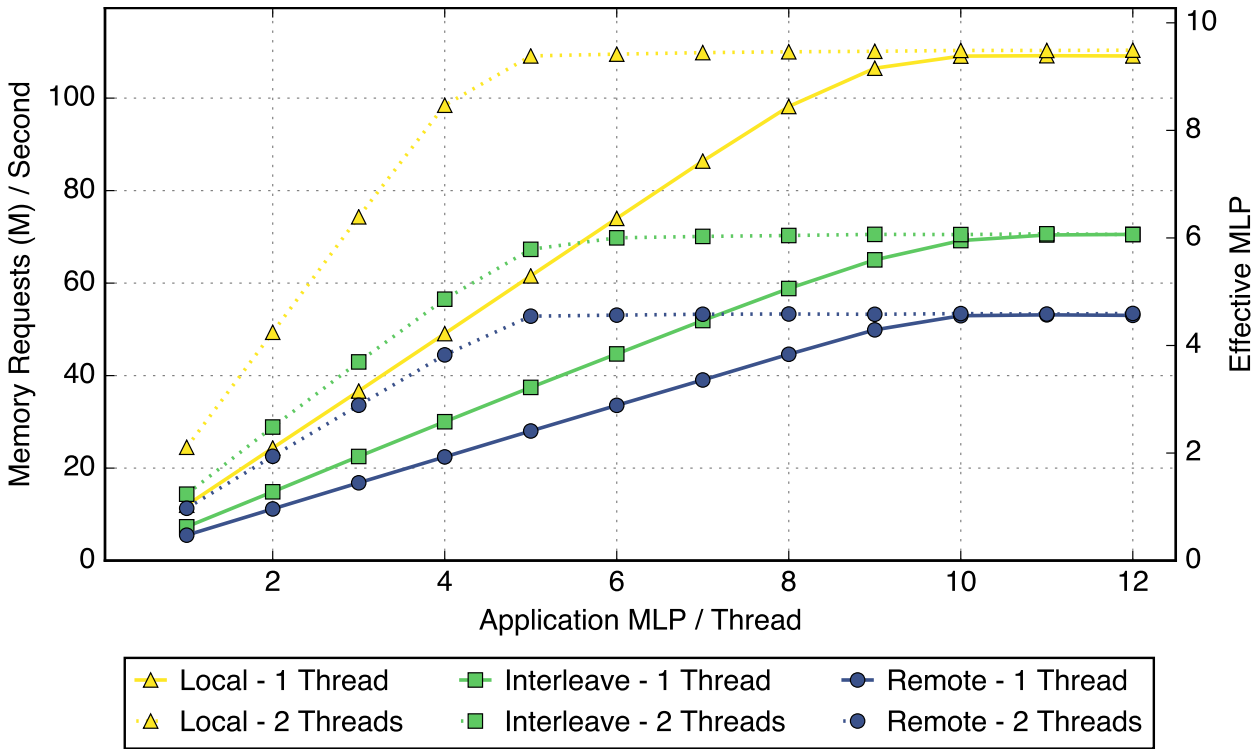

Figure 5.2: Memory bandwidth achieved by parallel pointer chase microbenchmark (random) in units of memory requests per second (left axis) or equivalent effective MLP (right axis) versus the number of parallel chases (application MLP). Single core using 1 or 2 threads and differing memory allocation locations (local, remote, and interleave).

After exploring how application MLP changes bandwidth (requirement 3) and how many outstanding misses the hardware supports (requirement 4), we now return to the impact of the instruction window size (requirement 2). Using inline assembly, we add `nop`s to our pointer-chase loop, thus moving the loads farther apart in the instruction stream. To examine the net result, we use the metric instructions per miss (IPM), which is the inverse of the common misses per kilo-instruction metric (MPKI = 1000/IPM).

As shown in Figure 5.3, window size is an important constraint on our platform, as bandwidth is inversely related to IPM, which confirms our intuition that memory requests must fit in the window in order to be issued. Assuming the loads are evenly spaced, we obtain a simple model for an upper-bound (with w as the window size):

$$MLP_{model} = \min(MLP_{max}, w/IPM + 1)$$

For our IVB core, the maximum memory bandwidth (MLP_{max}) is 10 and the instruction window size (w) is 168. The curved region adds one because if the window can hold n IPM-sized intervals, it can hold $n + 1$ endpoints. Our model is pessimistic as it assumes cache misses are evenly spaced. If there is substantial variation in the miss interval (jitter), it is

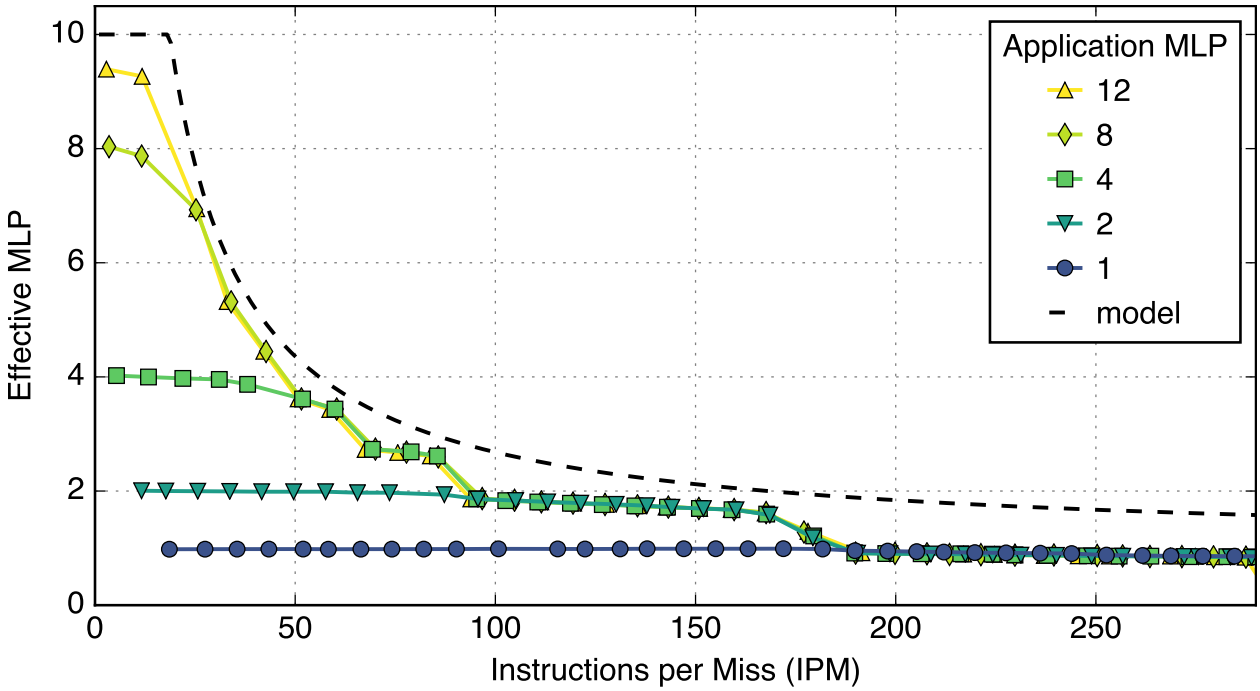

Figure 5.3: Memory bandwidth achieved by parallel pointer chase microbenchmark with varying number of nops inserted (varies IPM). Using a single thread with differing numbers of parallel chases (application MLP).

possible to exceed the model bound, but we find this simple model instructive for the rest of the study as we observe bandwidth is inversely related to IPM.

Memory bandwidth can also be constrained by frequent TLB misses. The four requirements above are necessary for a load to issue, but once issued, missing in the TLB incurs a latency penalty for its refill, which in turn will decrease bandwidth for the same number of outstanding memory requests. IVB's Linux distribution supports Transparent Huge Pages (THP) [117], which eagerly combines consecutive 4 KB pages into 2 MB pages when possible. IVB also supports 1 GB pages, but these must be set aside by Linux in advance and require substantial application code modifications. Larger pages not only reduce the chance of a TLB miss, but they also reduce the time per TLB refill by needing fewer hops to walk the page table and by reducing the size of the page table working set (better cache locality).

Figure 5.4 varies the page size (2 MB or 1 GB) and the array size (1 GB or 16 GB) for our pointer-chase synthetic microbenchmark. With 2 MB pages provided by THP, most loads for both array sizes will result in a cache miss and a TLB miss (IVB has only 32 2 MB TLB entries), but the maximum bandwidth obtained with the larger array is substantially reduced due to increases in TLB refill time (confirmed by performance counters). The TLB refill time increases due to worse cache locality during the page table walks, as a greater number of page table entries are needed to encompass the larger array. Using 1 GB pages restores the bandwidth because it reduces the TLB refill time by improving cache locality

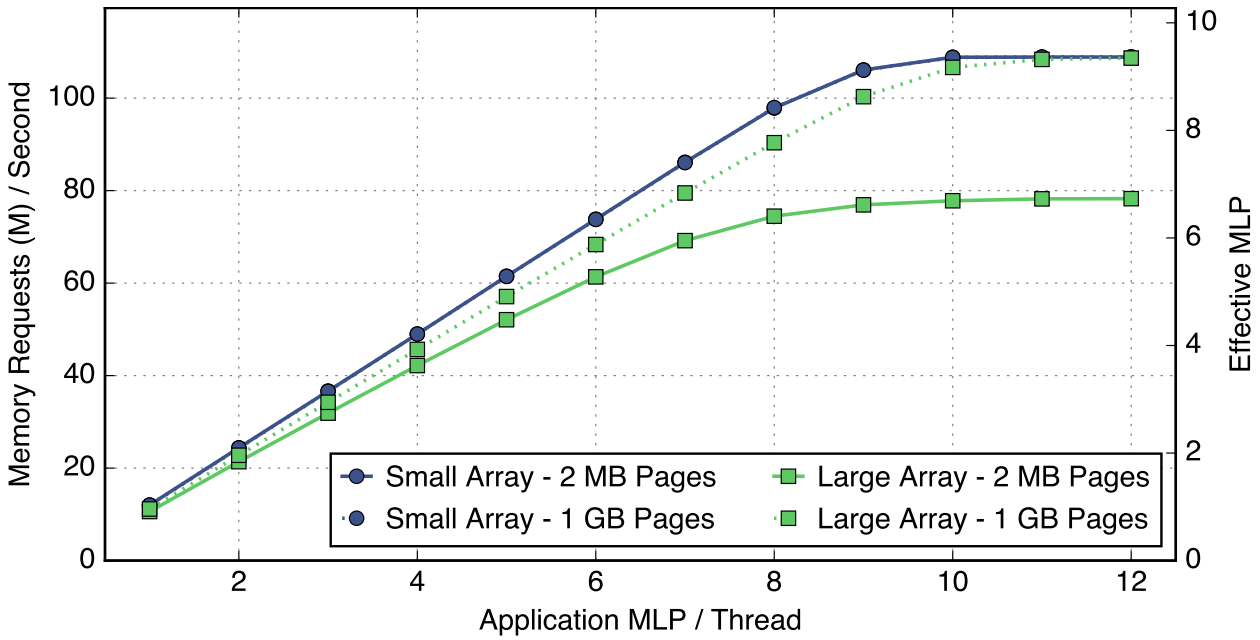

Figure 5.4: Impact of 2 MB and 1 GB page sizes on memory bandwidth achieved by single-thread parallel pointer chase for array sizes of *small* (1 GB) and *large* (16 GB).

during the page table walks. With 1 GB pages, the page table for the large array will need only 16 entries, and even though these entries will be frequently reloaded (IVB only supports 4 1 GB TLB entries), the entries will likely remain in the L1 cache.

Our random microbenchmark exemplifies the worst case for the TLB, so any form of locality will reduce the performance penalties from TLB misses. The difference also becomes more pronounced as application MLP increases because the increased concurrency increases the refill time as there is a limit to the number of simultaneous page table walks so the waiting time to start a walk will increase too.

We further parallelize our pointer-chase microbenchmark to see the bandwidth potential of the entire system (Figure 5.5). We achieve a maximum system throughput of 76 GB/s. When accessing remote memory, bandwidth is halved just as in the single-core case. A core is capable of using more than its fair share of bandwidth, as there are 8 cores per socket that are each capable of 10 misses outstanding, but the socket appears to be effectively capable of just over 50. Compared to the single-core case (Figure 5.2), the bandwidth saturates more gradually and requires oversubscription (application MLP > effective MLP) due to stochastic effects from queuing. When loading k banks, it is statistically unlikely that with only k random requests, there will be exactly one request per bank.

The data in this section shows the maximum achievable memory bandwidth for a core, socket, or entire system given the amount of application MLP, IPM, memory location (NUMA), and page size. In the following section, we measure the memory bandwidths achieved by the high-performance graph codebases.

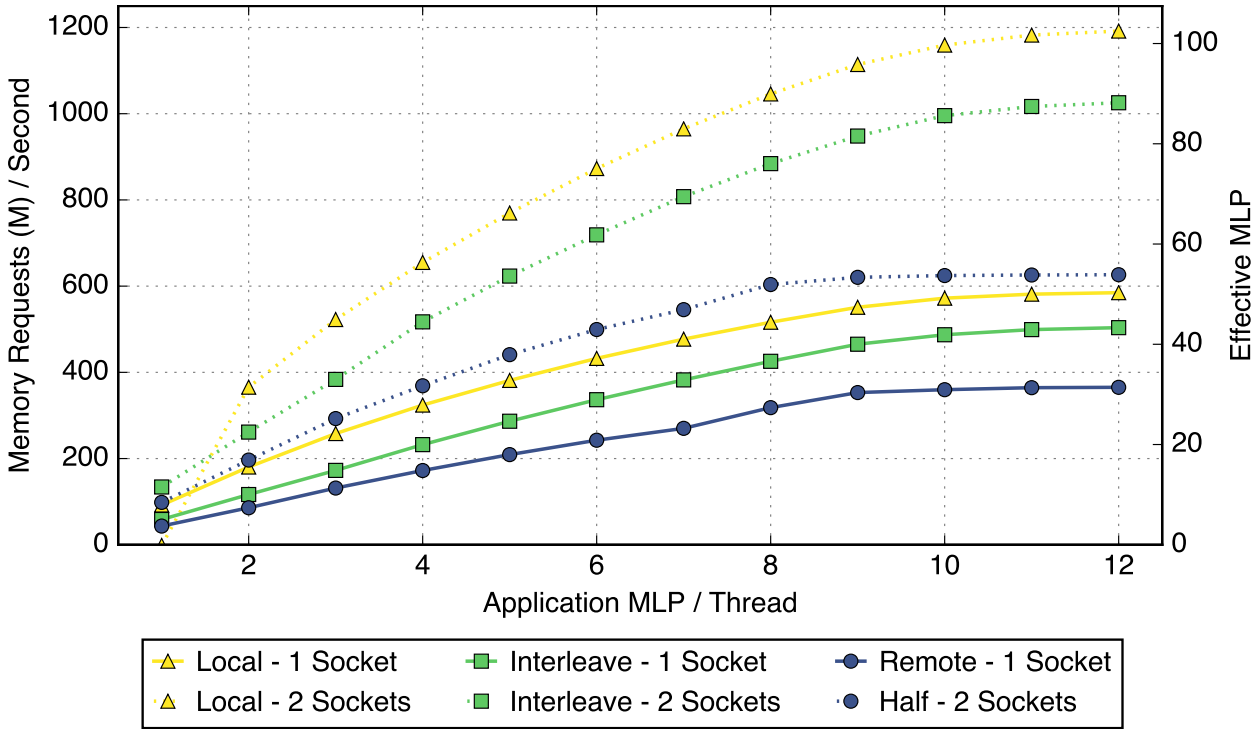

Figure 5.5: Random bandwidth of a socket or the whole system with different memory allocations.

5.4 Single-Core Analysis

In this section, we begin to characterize our workload using only a single thread on a single core in order to remove any parallel execution effects (multithreading, poor parallel scaling, load imbalance, and NUMA penalties). Despite being amongst the highest-performance implementations, all three codebases often execute instructions at a surprisingly low IPC (Figure 5.6), and this disappointing performance observed is not specific to any graph algorithm or codebase. The input graph does have a large impact as we will discuss later in this section.

Figure 5.6 shows that there is an unsurprising tradeoff between computation and communication, as no executions sustain both a high IPC and a high memory bandwidth. A processor can only execute instructions at a high rate if it rarely waits on memory, and hence consumes little memory bandwidth. Conversely, for a processor to use a great deal of memory bandwidth, it must have many memory requests outstanding, causing it to be commonly waiting on memory and will thus execute instructions slowly. Although some executions do use an appreciable amount of compute (upper left of Figure 5.6) or use an appreciable fraction of the memory bandwidth (lower right), most do not. Many executions are actually in the worst lower-left quadrant, where they use little memory bandwidth, but their compute throughput is also low, presumably due to memory latency.

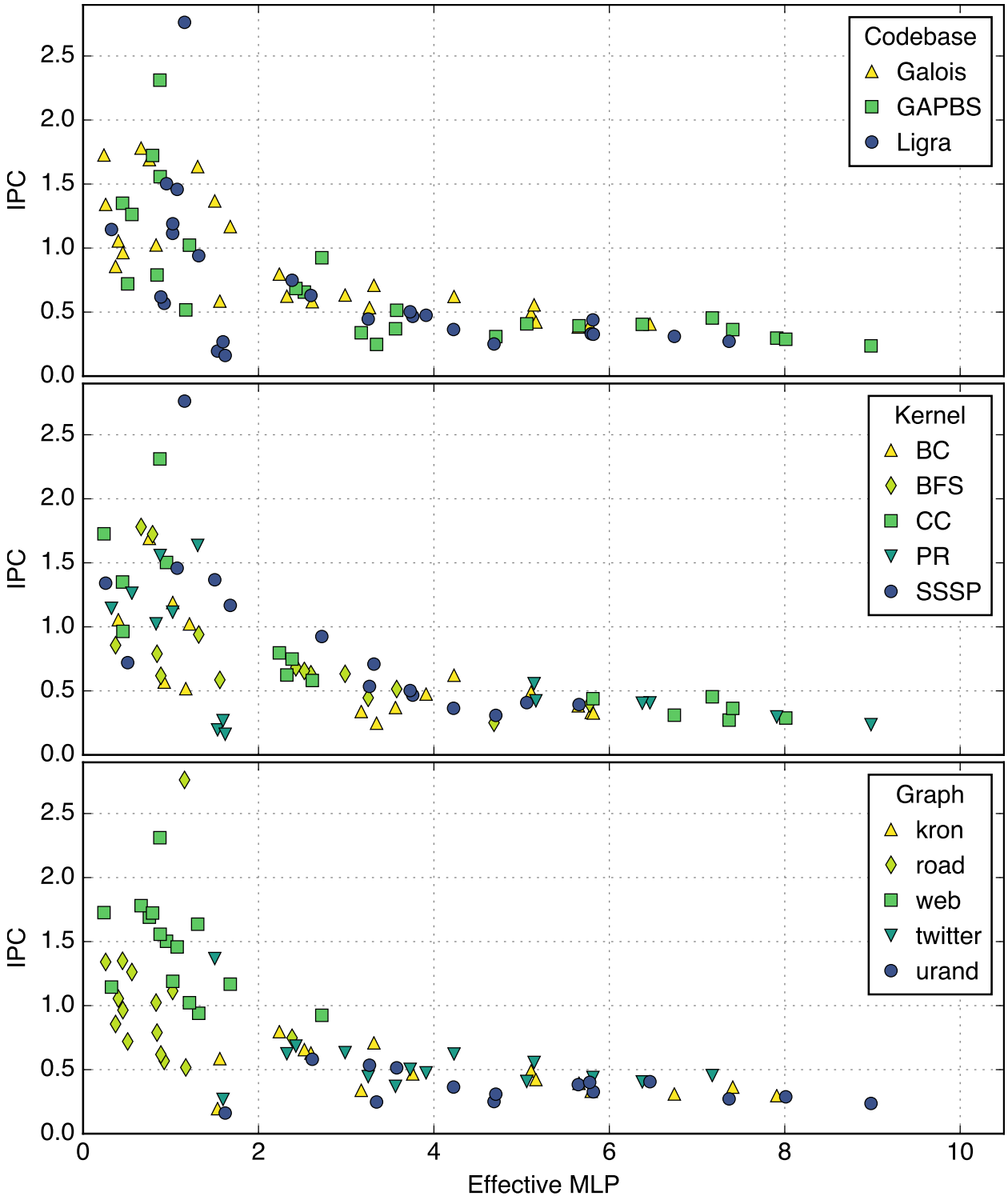

Figure 5.6: Single-thread performance in terms of instructions per cycle (IPC) of full workload colored by: codebase (top), kernel (middle), and input graph (bottom).

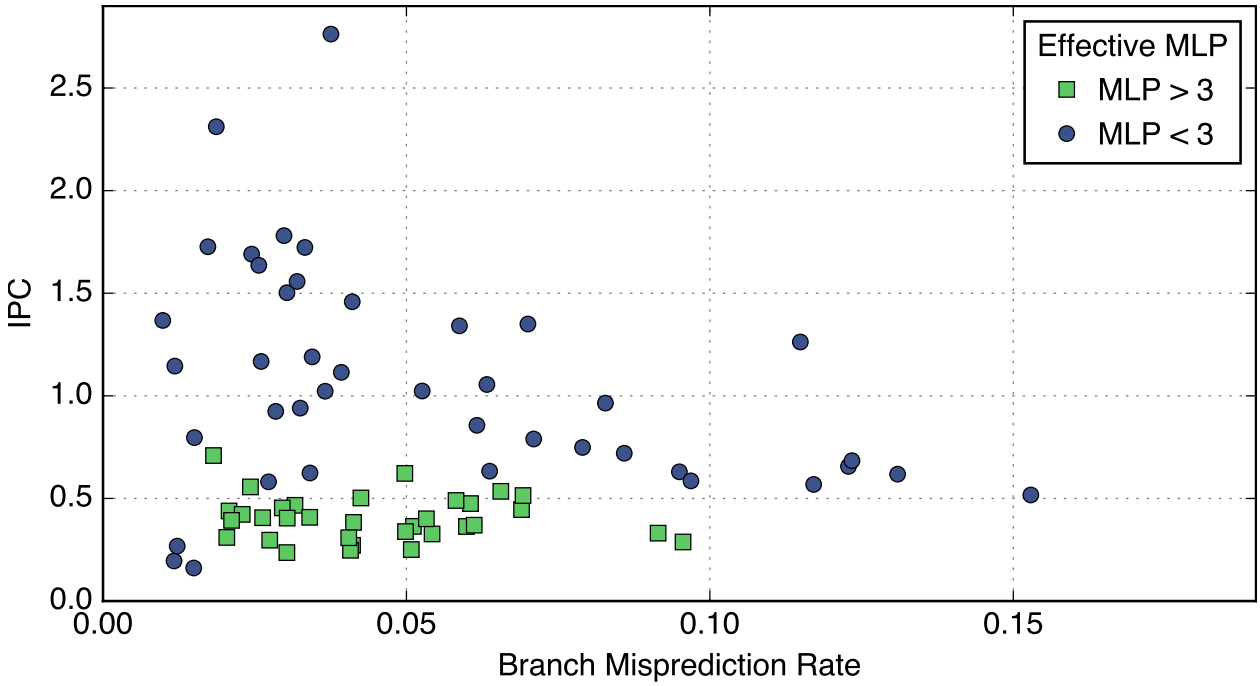

Figure 5.7: Single-thread performance of full workload relative to branch misprediction rate colored by memory bandwidth utilization.

In general across our codebases, kernels, and inputs graphs, a single core struggles to use all of the raw bandwidth available (10 outstanding misses). With the same communication volume, utilizing more bandwidth should lead to higher performance. Using the four requirements from Section 5.3, we investigate what is limiting the core's bandwidth utilization for what should be a memory-bound graph processing workload.

To have many loads outstanding, the processor must first fetch those load instructions, and this typically requires correctly predicting the control flow. Although frequent branch mispredictions will be harmful to performance in theory, if the processor is already waiting on memory (achieving moderate memory bandwidth utilization), performance is insensitive to the branch misprediction rate (Figure 5.7), implying many of these branches are miss independent. When the processor is not memory-bound, frequent branch mispredictions will hurt performance, but a low misprediction rate is no guarantee for good performance, implying there are other bottlenecks.

Once the processor fetches the future outstanding loads, those loads need to be able to fit into the instruction window, and the model from Section 5.3 serves as an upper bound for our workload (Figure 5.8). Although the model is technically a pessimistic upper bound since it assumes outstanding loads are evenly spaced apart, in practice this seems to be a suitable approximation. In spite of the core being capable of handling 10 outstanding misses, an IPM of greater than 18.7 will not allow all these loads to fit in the window according to our model. Most of the executions have an IPM greater than this cutoff, and thus have their

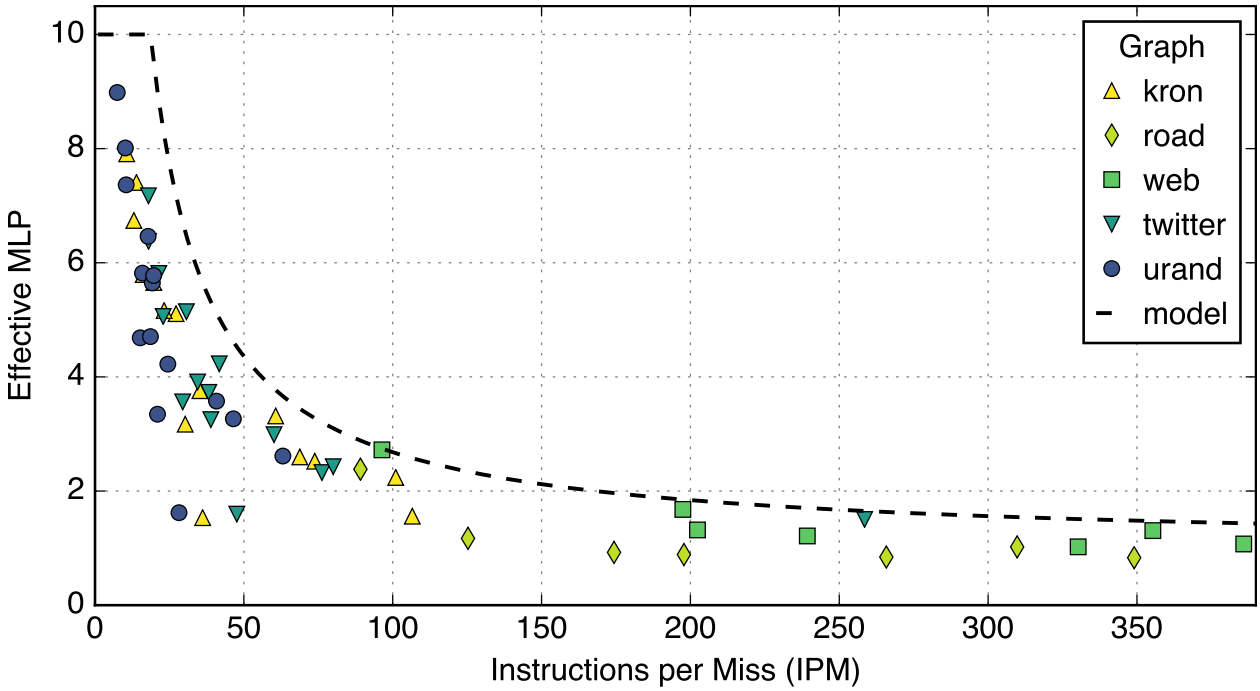

Figure 5.8: Single-thread achieved memory bandwidth of full workload relative to instructions per miss (IPM). *Note: Some points from road & web not visible due to IPM>1000 but model continues to serve as an upper bound.*

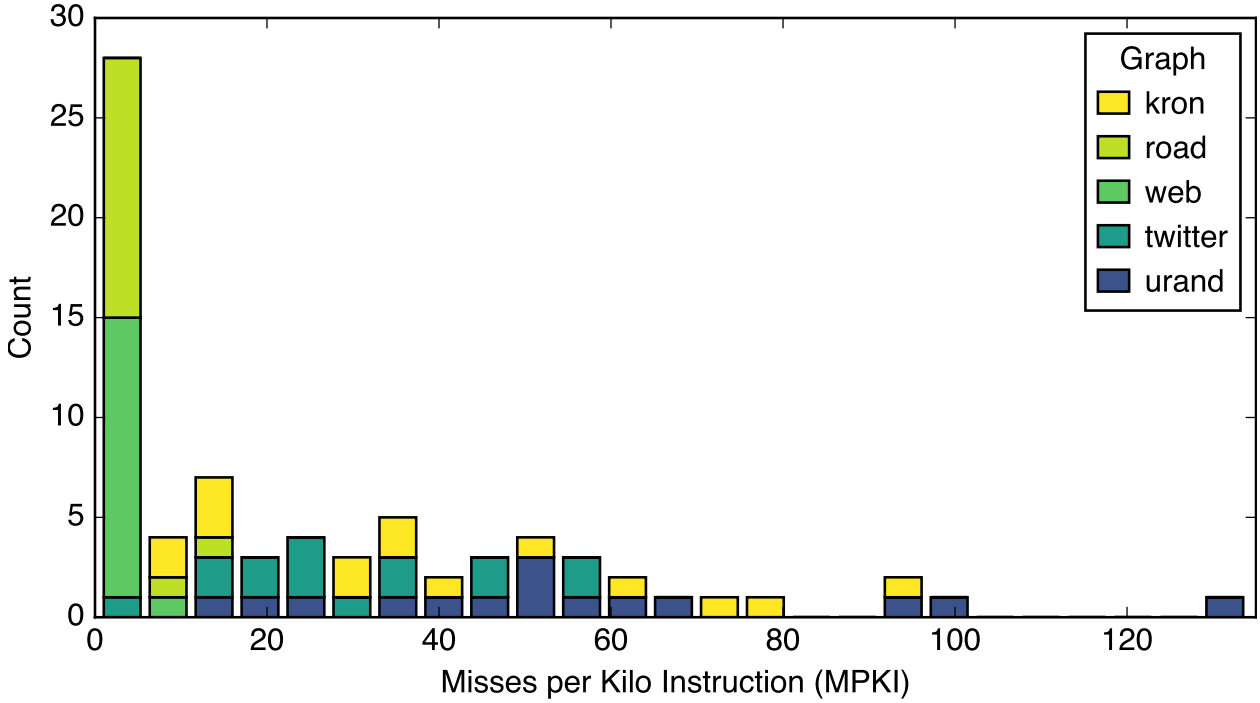

Figure 5.9: Histogram of MPKI (in terms of LLC misses) of full workload specified in Section 5.2 executing on a single thread. Most executions have great locality (low MPKI), especially those processing the web or road input graphs.

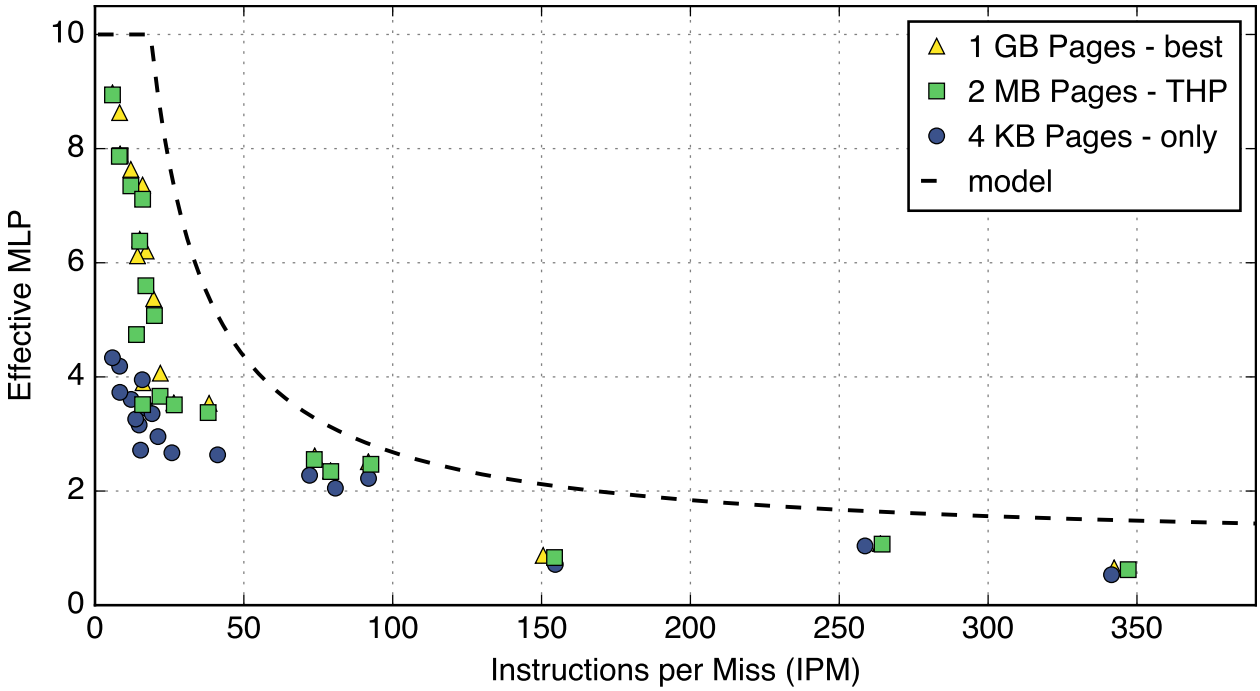

Figure 5.10: Single-thread achieved memory bandwidth of GAPBS for all kernels and graphs varying the operating system page size. *2 MB Pages - THP* uses Transparent Hugepages (THP) and lets the operating system choose to promote 4 KB to 2 MB pages (happens frequently). *1 GB Pages - best* is the fastest execution using manually allocated 1 GB pages for the output array, the graph, or both.

effective bandwidth limited by the instruction window size. The caches achieve a modest hit rate (Figure 5.9), which raises the IPM by absorbing much of the memory traffic.

As mentioned above, the properties of the graph can have a substantial impact on the cache performance, which in turn will affect not only the amount of memory traffic, but also how fast it can be transferred. For example, in Figure 5.8 the graph road has a high IPM because it is much smaller than the other graphs. The topology can also have an impact, as the graphs kron and urand are about the same size and diameter, and yet urand typically uses more bandwidth because it has a lower IPM caused by more cache misses. The graph kron experiences fewer cache misses because it is scale-free, as a few high degree vertices will be accessed frequently (great temporal locality). Finally, the graph web has a higher degree, which allows for longer contiguous reads (better spatial locality) causing more cache hits and thus a higher IPM.

Although there is not typically substantial benefit from using 1 GB pages, using 4 KB pages does have quite a performance penalty. Fortunately, THP is on by default and requires no application modifications. We vary the operating system page size for the GAPBS codebase in Figure 5.10. Relative to the baseline using THP (2 MB pages whenever possible), using 1 GB pages improves performance by more than 10% in only 4/25 cases but disabling

THP, which forces all pages to be 4 KB, decreases performance by at least 10% in 19/25 cases. To use 1 GB pages, we modify GAPBS to allocate 1 GB pages for the graph, the output array, or both (typically the best) and pick whichever one is fastest. The general insensitivity to the 1 GB page size for our graph workload is another indication of locality.

We compare data dependencies versus branch mispredictions to explain performance slowdown, and while difficult to disentangle, the evidence points much more strongly to the former than to the latter. With a combination of knowledge of IVB's architecture and confirmation from performance counters, we eliminate other possible performance limiters. Due to sophisticated hashing of memory addresses, there is no significant bank contention in the LLC or at the memory controllers. The load buffer can hold 64 entries, so it rarely limits outstanding loads before the ROB (168 entries) or the MSHRs (10 per core). Mis-speculated loads are already counted by the performance counters we utilize. The graph workloads we measure have clearly dominant application phases (no substantial temporal variation).

None of the executions of actual graph processing workloads are able to achieve a memory bandwidth corresponding to the 10 outstanding misses our synthetic microbenchmarks demonstrate the cores are capable of sustaining, and most are not even close. For a single thread, the biggest bandwidth limiter is fitting loads into the instruction window, which prevents off-chip memory bandwidth from becoming a bottleneck.

5.5 Parallel Performance

With an understanding of the limits and capabilities of a single thread, we move on to the whole system. Running the codebases at full capacity delivers speedups for all executions, and with 32 threads on 16 cores we achieve a speedup greater than 8× (relative to single-thread) in 49 of 75 cases and a median speedup of 9.3× (Figure 5.11). Unfortunately, some of the executions (typically road and web) increase their bandwidth consumption by more than they improve runtime, implying their parallel executions have more memory traffic than their single-threaded counterparts.

The compute and throughput utilization for the parallel executions (Figure 5.12) is strikingly similar to utilizations for a single core (Figure 5.6). Although web and sometimes road break the trend by simultaneously using more compute throughput and memory bandwidth, they do move extra data. The similarities between parallel utilization and serial utilization suggest that the bottlenecks of the core persist and hurt utilization at the system scale. Due to the generally linear relation between performance and memory bandwidth, fully utilizing the off-chip memory system could improve performance by 1.3–47× (median 2.4×).

There may be graph algorithm implementations with better parallel scaling properties, but the advanced algorithms were used in this chapter because they deliver better absolute performance. Parallel scaling can be hampered by software issues (poor scalability, load imbalance, synchronization overheads, and redundant communication), but in the remainder of this work we will consider hardware imposed complications for parallelization: NUMA and multithreading.

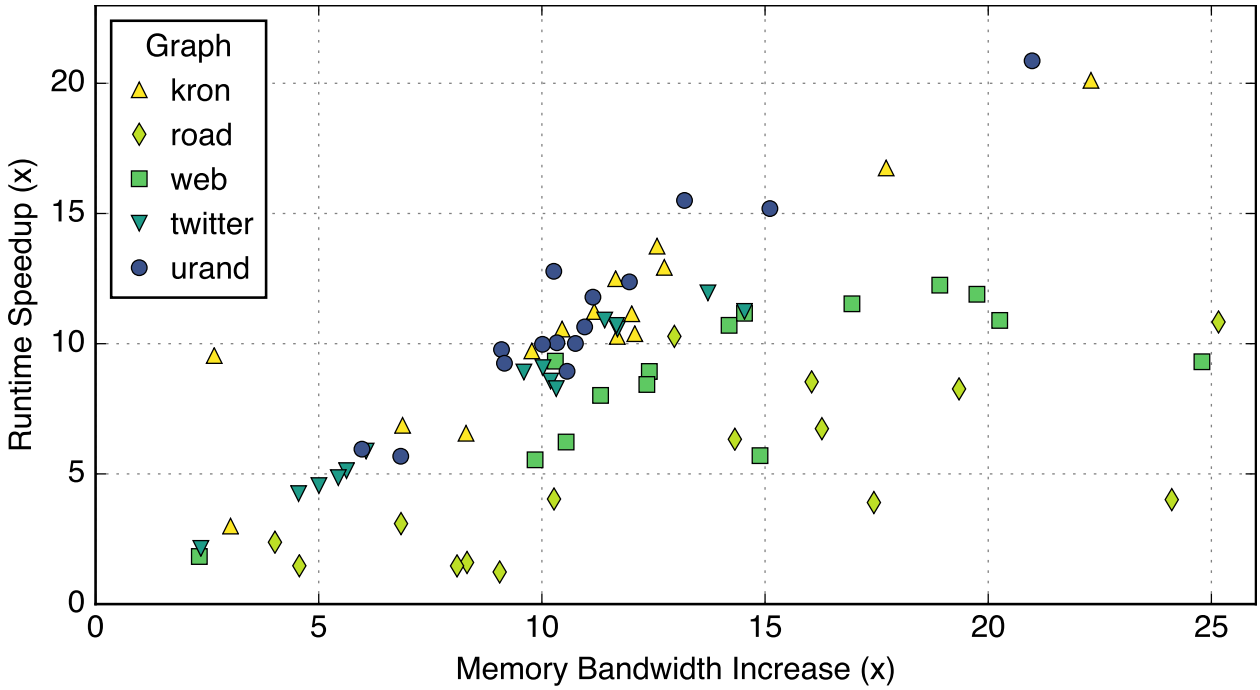

Figure 5.11: Improvements in runtime and memory bandwidth utilization of full workload for full system (32 threads on 16 cores) relative to single thread performance.

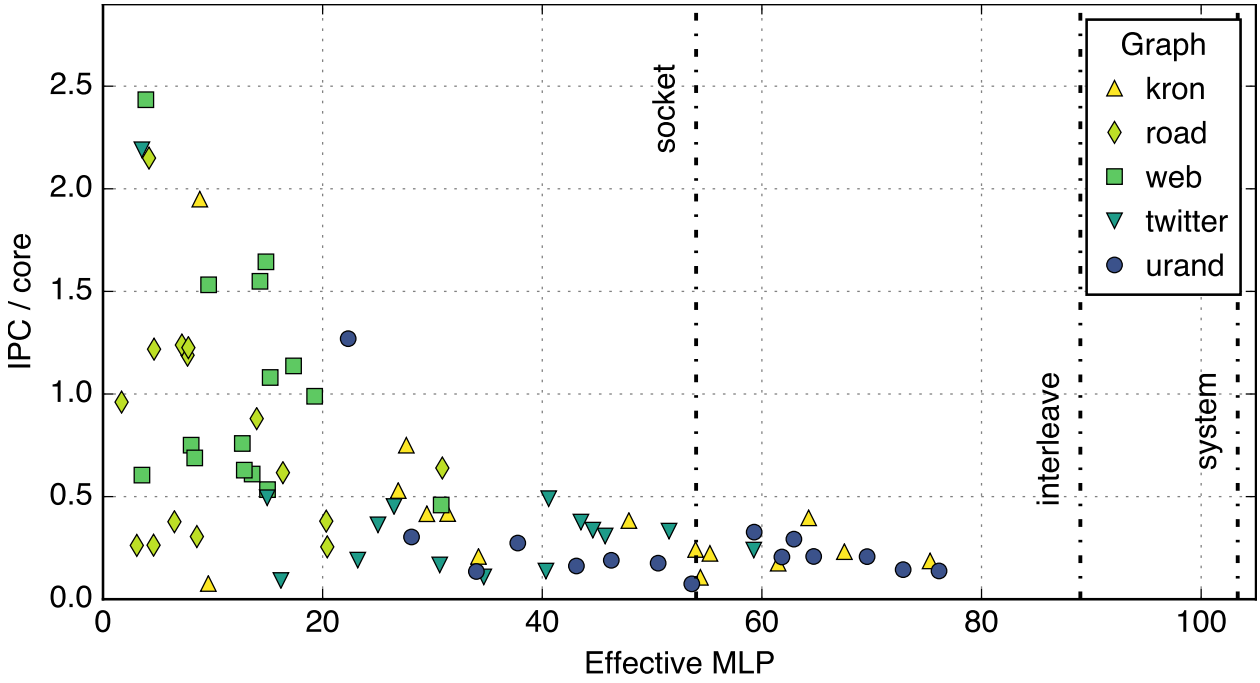

Figure 5.12: Full system (32 threads on 16 cores) performance of full workload. Vertical lines are maximum achieved bandwidths (Section 5.3) for a single socket (socket), both sockets with interleaved memory (interleave), and both sockets with local memory (system).

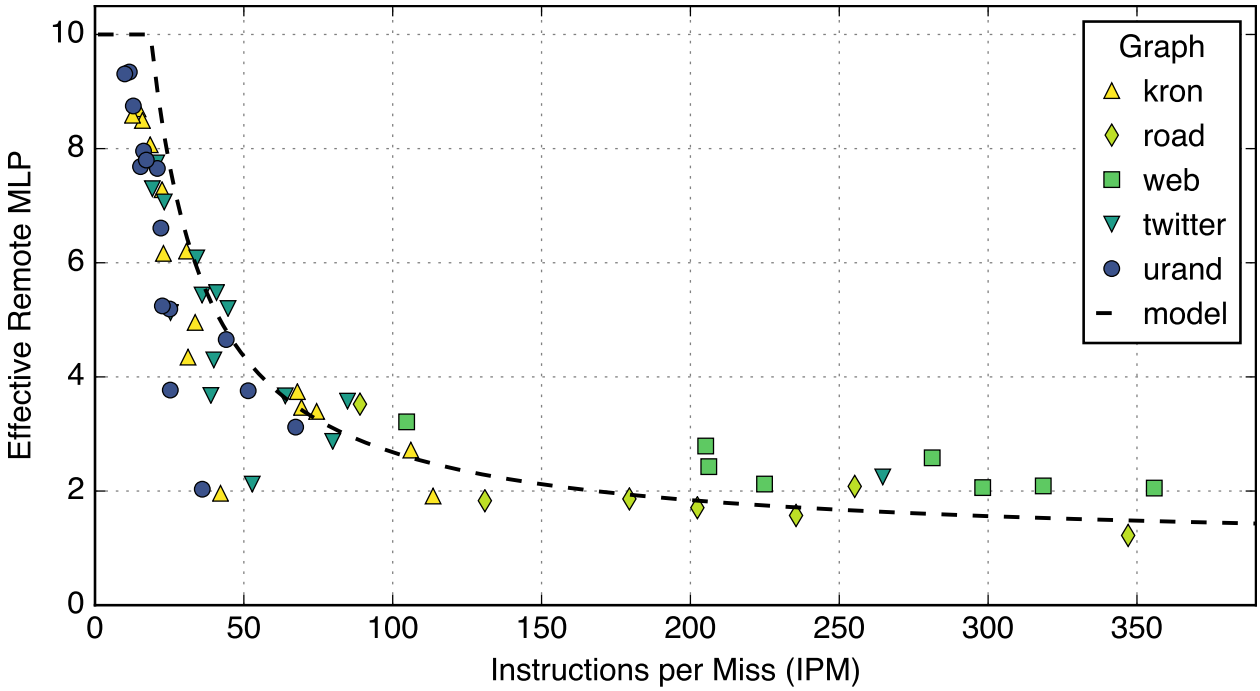

Figure 5.13: Single-thread achieved memory bandwidth of full workload executing out of remote memory. Calculating effective MLP with remote memory latency (instead of local memory latency) returns a result similar to local memory results (Figure 5.8).

5.6 NUMA Penalty

With multi-socket systems, non-uniform memory access (NUMA) penalties are a common challenge. From the results of Section 5.3, it would appear that NUMA should halve performance, but our results indicate the penalty for NUMA may be substantially less severe in practice.

For a single thread using only remote memory, performance is halved as it transfers the same amount of data with the same number of outstanding memory requests but at twice the latency for effectively half the bandwidth. Calculating the effective MLP with the remote memory latency instead of the local memory latency shows the workload still obeys the simple bandwidth model (Figure 5.13).

With more cores, this NUMA penalty is reduced (Figure 5.14), and for executions that use less memory bandwidth (higher IPM), the NUMA penalty is reduced further. A core using only remote memory is clearly an adversarial worst case. For a full system workload without locality, half of the traffic should still go to local memory. Consequently, the *interleaved* pattern in Figure 5.14 is more realistic and it has one third the performance loss of *remote* (median 1.16× slowdown vs. 1.48× slowdown).

We confirm that NUMA has a moderate performance penalty. Unfortunately, many graphs of interest are low diameter and hard to partition effectively [63], so it is challenging

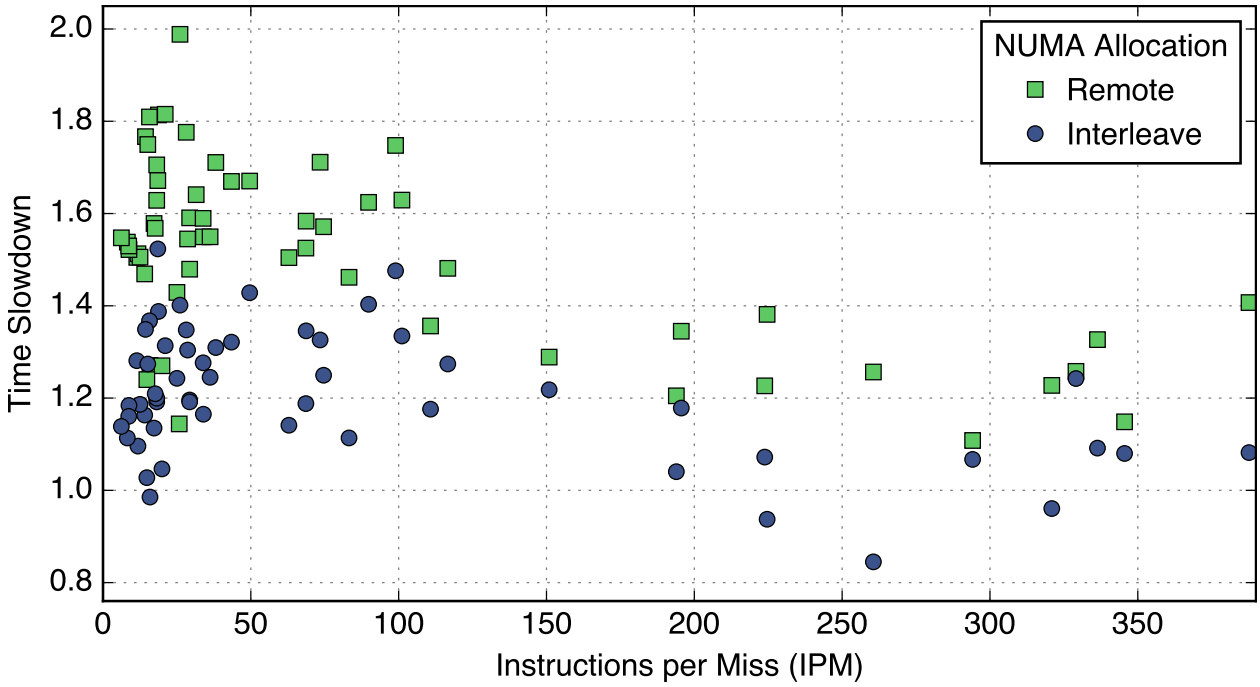

Figure 5.14: Full workload slowdowns for single-socket (8 cores) executing out of remote memory or interleaved memory relative to executing out of local memory.

to avoid inter-socket communication. Therefore, efforts to move computation (rather than data) have fared the best when optimizing graph processing for NUMA [2, 40].

5.7 Limited Room for SMT

Multithreading, and in this work's context of a superscalar out-of-order processor, simultaneous multithreading (SMT) [56], aims to increase utilization. The additional software-exposed parallelism threads provide can be used to mitigate unresolved data dependences by increasing application MLP as well as reducing the demand placed on branch prediction since each thread will have fewer instructions in flight. Using IVB, we measure the performance gains of using a second thread per core, which evaluates how well SMT reduces the performance loss from unresolved data dependencies and branch predictions without incurring new overheads. On IVB, all SMT threads on a core share the same instruction window, so multithreading will be unable to ameliorate the memory bandwidth bottleneck induced by the instruction window size that we observe to limit performance in the rest of this work.

Across all scales (single core, single socket, or single system), the second thread is usually beneficial, but only to a modest degree (Figure 5.15) as most speedups are less than 1.5×. As more cores are used, multithreading becomes less beneficial (median 1.26× for 1 core versus median 1.12× for two sockets). This is unsurprising since at the socket and system

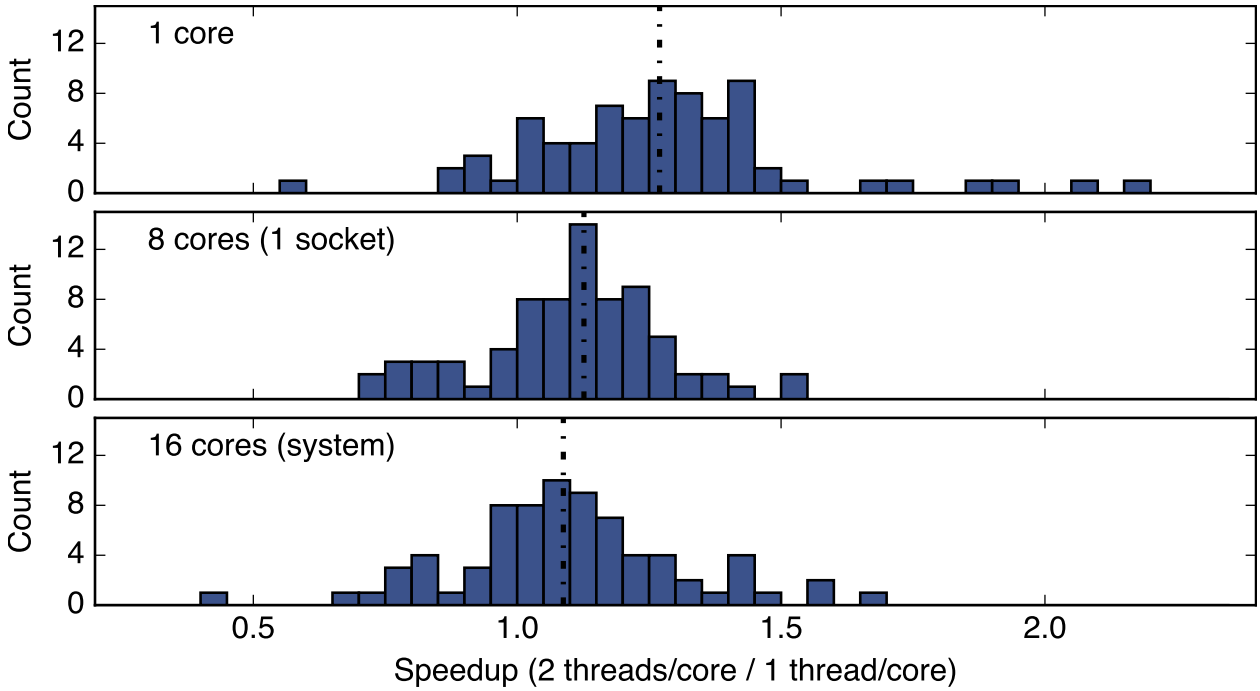

Figure 5.15: Distribution of speedups of using two threads per core relative to one thread per core of full workload for one core, one socket (8 cores), and whole system (2 sockets). Dotted line is median.

level, parallel performance challenges (load imbalance, synchronization overhead, and NUMA penalties) also limit performance and they make what multithreading aims to ameliorate (unresolved data dependencies and branch misprediction penalties) a comparatively smaller fraction of the performance limiters. Even so, these modest speedups from SMT are not inconsequential, as SMT economically improves system performance.

Multithreading also has the potential to introduce new performance challenges. More threads increase parallelism, which in turn can worsen the damage caused by load imbalances and synchronization overheads. Worse yet, more threads can compete for capacity in the cache resulting in increased memory traffic. Analogous to the results for parallel execution (Section 5.5), the road and web graphs in Figure 5.16 are examples of this competition as the improvement in bandwidth is greater than the improvement in runtime.

For a single thread, we find the biggest performance limiter to be fitting loads into the instruction window, and SMT is no different as the addition of a second thread to the same core still mostly obeys our simple model since it shares the same window (Figure 5.17). If the workload of the two threads is heterogenous it is possible for an SMT core to exceed our simple model. One thread could generate most of the cache misses sustaining a high effective MLP while the other thread (unencumbered by cache misses) could execute instructions quickly to increase IPM. In practice, the variation between threads is modest and thus most measured results are not far above our model.

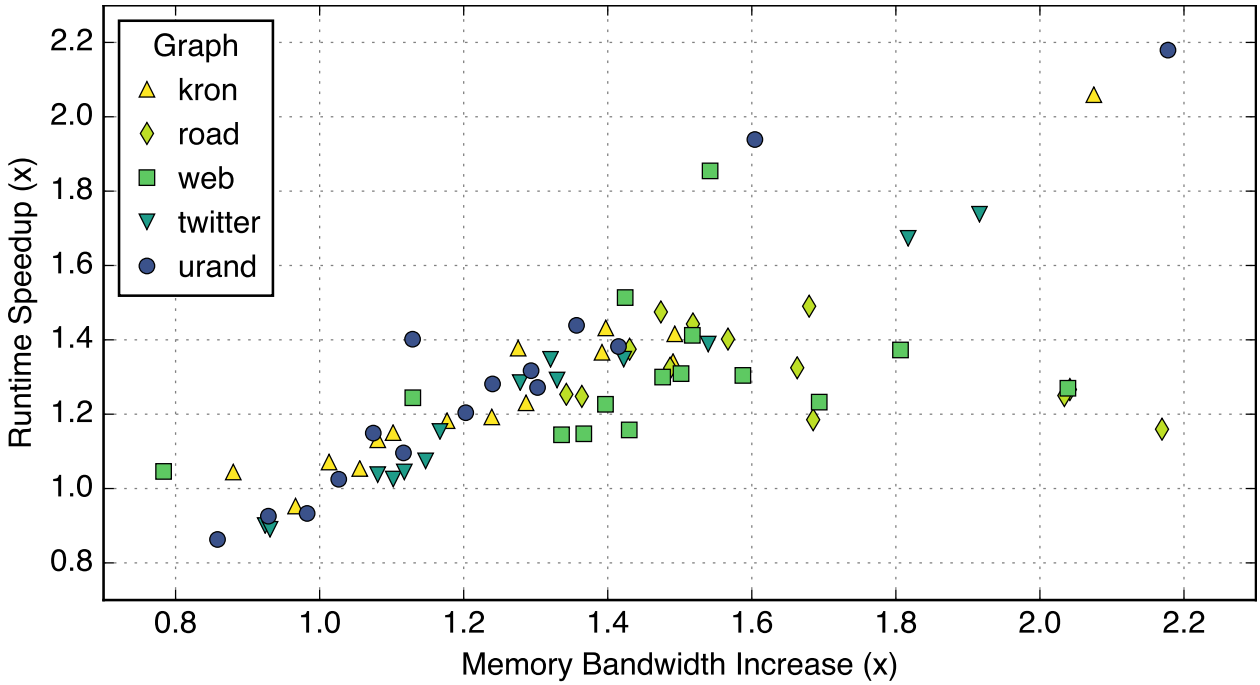

Figure 5.16: Improvements in runtime and memory bandwidth utilization of full workload for one core using two threads relative to one thread.

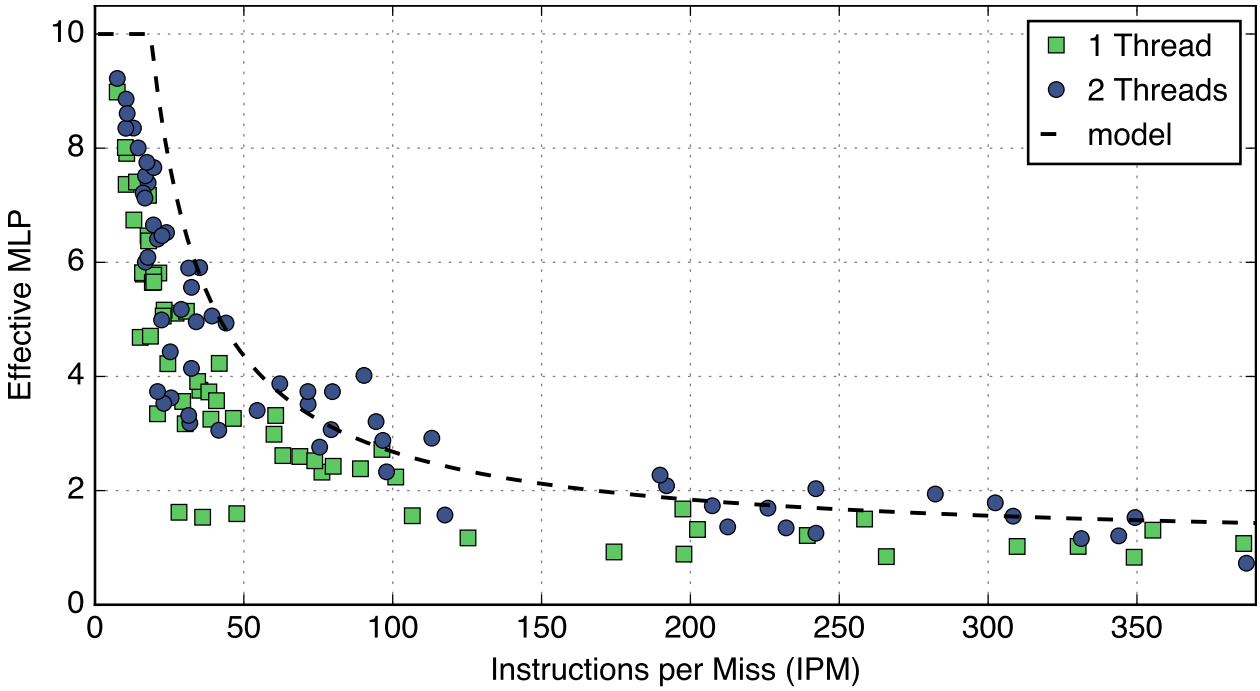

Figure 5.17: Achieved memory bandwidth of full workload relative to instructions per miss (IPM) with one or two threads on one core.

Multithreading can improve performance, but in the context of this study (graph processing workload on a superscalar out-of-order multi-socket system), it has limited potential. The modest improvements two-way multithreading provides in this study cast doubt on how much more performance is to be gained by additional threads.

5.8 Related Work

Our study touches on many aspects of computer architecture, so we focus this section specifically on prior work relevant to graph algorithms. Compared to our prior publication [19], this chapter goes into additional detail throughout. Compared to prior work on the architectural requirements for graph algorithms, our study has a much larger and more diverse graph workload. We study 5 kernels from 3 codebases with 5 input graphs (some of which are real and not synthetic).

A survey [98] of both hardware and software concerns for parallel graph processing lists "poor locality" as one of its chief concerns. Although it is cognizant of the greater cost of heavily multithreaded systems, it argues they are better for graph algorithms due to their memory latency tolerance and support for fine-grained dynamic threading. Bader et al. [11] also endorse heavily threaded systems because of concerns of memory accesses being mostly non-contiguous (low locality).

Cong et al. [43] compare a Sun Niagara 2 to a IBM Power 7 when executing graph algorithms to understand architectural implications. Both platforms are multithreaded, but the Niagara 2 is in-order with 8-way threading and the Power 7 is out-of-order with 4-way threading. They find memory latency (not memory bandwidth) to be a bottleneck for both platforms, and neither platform has enough threads to fully hide it. Additionally, they find algorithmic tricks to increase locality that are beneficial for both platforms. They also present analytical models to explain the stochastic challenges of keeping a multithreaded pipeline fully utilized.

To better understand graph algorithm architectural requirements, prior work has explicitly examined the locality behavior of graph algorithms. Cong et al. [44] study several Minimum Spanning Tree algorithms with a reuse distance metric (temporal locality). They find graph algorithms do have less (but not no) locality, but observe some algorithms with less locality sometimes perform better, and hypothesize this is due to not accounting for spatial locality. Analytical models for BFS can accurately predict the reuse distance of BFS on certain random graphs [175]. Murphy et al. [114] examine serial traces from a variety of benchmark suites including graph algorithms. Despite locality metrics based on an extremely small cache for the time of publication, they observe that integer applications tend to have less locality than floating-point applications, but are still far better than random.

Efforts to improve performance by explicit NUMA optimizations typically require complicated manual modifications and are not generally applicable to all graph algorithms. Agarwal et al. [2] improve BFS performance using custom inter-socket queues. With a high-end quad-socket server, they are able to outperform a Cray XMT. Chhugani et al. [40] minimize

inter-socket communication for BFS, and provide a detailed performance model for their implementation.

Although hardware prefetchers may struggle to predict non-streaming memory accesses, explicit software prefetching has been investigated as a means to improve graph algorithm performance [2, 43, 77]. Not unlike explicit NUMA optimizations, for graph algorithms, using software prefetching requires human intervention. Software prefetching can be difficult to implement effectively for all graph algorithms because it is often hard to generate the addresses desired sufficiently before they are needed.

Green et al. investigate improving graph algorithm performance by reducing branch mispredictions using conditional moves [68]. They conclude that branch mispredictions are responsible for a 30%–50% performance loss, but in our results (Section 5.4) we do not observe such a large penalty when considering the limitations imposed by data dependences and fitting loads into the instruction window.

Runahead execution is a technique to improve processor performance in the presence of cache misses [53], and in the case of an out-of-order core, runahead execution attempts to economically obtain the benefits of a larger instruction window [115]. Rather than stalling the instruction window on a miss, runahead checkpoints and attempts to go forward. Much of the work after the checkpoint will have to be redone when the data for the original load returns, but hopefully by being able to move forward more cache misses will be exposed, thus leading to better bandwidth utilization. Van Craeynest et al. choose to not enter runahead mode to reduce the number of re-executed instructions if their predictor guesses runahead execution will be unable to expose more cache misses [156].

Prior work on SMT performance also casts doubt on how much performance remains to be exploited by SMT on large out-of-order processors. Raasch et al. investigate resource allocation within SMT processors and they conclude storage resources such as the instruction window should be statically partitioned while execution resources should be dynamically partitioned [135]. Their reasoning is that storage resources tend be held longer (e.g. instruction window blocked on a cache miss), so static partitioning helps prevent starvation. The long occupancy of storage resources gets at the crux of the problem of using SMT to improve the performance of out-of-order processors. SMT is ideal for multiplexing execution units to fill idle slots, but it can only subdivide storage resources. Since these storage resources are held for so long, there are no "idle slots" to fill. SMT provides a bigger performance boost for in-order processors than out-of-order processors because in-order processors have less of these contested storage resources [121]. Hily et al. demonstrate the result of this by showing that a large in-order SMT processor is able to achieve much of the performance of a large out-of-order processor while using substantially less resources [72].

The Cray XMT is a system explicitly designed to handle irregular problems including graph algorithms [154]. Designed for workloads without locality, it feature many hardware threads and no data caches. When considering its architecture, it is worth keeping in mind its ancestry, which going back in time includes the MTA-2 [6], the Tera [5], and the HEP [150]. The MTA-2 has full bisection bandwidth but no caches, and it uses heavy multithreading to tolerate memory latency. The XMT was intended to be cheaper than the MTA-2 by

Publication	Random Benchmark	Random Performance	Graph Algorithm	Graph Performance
Bader [11]	Random List Ranking	35.0×	CC	5.5×
Madduri [101]	GUPS	2.8×	BC	1.8×
Nelson [118]	GUPS	2.2×	BFS	1.6×
	GUPS	2.2×	PR	4.4×

Table 5.3: Improvement for specialized platform over baseline platform for random and graph algorithm benchmarks. Random benchmarks (e.g. GUPS) are poor predictors of graph algorithm performance and often underestimate graph algorithm performance of baseline platform.

leveraging an interconnect originally developed for other Cray systems [154]. This repurposed network hinders scalability because it does not provide full bisection bandwidth, so small caches were added to the memory controllers (no replication so need for coherence) to increase the effective memory bandwidth. The acknowledgment that caches were expected to provide performance improvement for graph algorithms demonstrates exploitable locality within graph algorithms [154].

There has been substantial effort characterizing graph processing workloads on GPUs. Since GPUs are optimized for regular data parallelism, Burtscher et al. propose metrics to quantify control-flow irregularity and memory-access irregularity and they perform performance counter measurements on real hardware [34]. For some graph algorithms, they observe the performance characteristics depend substantially on the inputs. A continuation of that research uses a software simulator to change GPU architectural parameters and observes performance is more sensitive to L2 cache parameters than to DRAM parameters, which suggests there is exploitable locality [125]. Xu et al. also use a simulator and identify synchronization with the CPU (kernel invocations and data transfers) as well as GPU memory latency to be the biggest performance bottlenecks [166]. Che et al. profile the Pannotia suite of graph algorithms and observe substantial diversity across algorithms and inputs [37]. Wu et al. investigate the most important primitives needed for higher-level programming models for graph algorithms [164]. Contrasting these GPU works from our work, in addition to the difference in hardware platform (CPU versus GPU), we use much larger input graphs enabled by executing on real hardware (no downsizing to reduce simulation time) and by using server-sized memory (not constrained by GPU memory capacity).

Some prior work compares graph algorithm performance with random benchmarks. The Giga Updates per Second (GUPS) metric, also known as the RandomAccess within the HPC Challenge Benchmark [52], measures a system's ability to perform updates to randomly generated locations. GUPS is not publicly stated to be representative of graph algorithms, and yet some still use it when comparing a specialized platform to a baseline for graph processing. Comparing the results of random bandwidth focused custom graph platforms with cache-centric conventional baselines, we observe that the random metrics are a poor

predictors of graph algorithm performance in practice (Table 5.3). The random benchmarks frequently overestimate the benefit of the specialized hardware because some algorithms may not scale perfectly or there is exploitable locality within them.

5.9 Conclusion

Our diverse workload (varied implementations, algorithms, and input graphs) demonstrates there is no single representative benchmark and we find the input graph to have the largest impact on the performance characteristics.

Most of our workload fails to fully utilize IVB's off-chip memory bandwidth due to having an insufficient number of outstanding memory requests. The biggest bandwidth bottleneck is the instruction window, because it cannot hold a sufficient number of instructions to incorporate the needed number of rare cache-missing instructions. A high LLC hit rate makes these cache misses rare, and we find this challenges the misconception that graph algorithms have little locality. TLB misses are only measurably detrimental when at least a moderate amount of memory bandwidth is utilized, and we find transparent huge pages to be effective at ameliorating much of the performance loss due to TLB misses. Branch mispredictions and unresolved data dependences can also hinder memory bandwidth utilization, but they are secondary to the interaction between the cache hit rate and the instruction window size. Bandwidth is also moderately hindered by NUMA effects, so software techniques to increase intra-socket locality or hardware techniques to decrease inter-socket latency will be beneficial.

The parallel scaling of our workload indicates that performance typically scales linearly with memory bandwidth consumption. Since our workload fails to fully utilize IVB's memory bandwidth, an improved processor architecture could use the same memory system but improve performance by utilizing more memory bandwidth. For our workload on IVB, SMT is typically beneficial, and when it improves performance, it does so by using more memory bandwidth. Unfortunately, in the context of an out-of-order core, SMT helps only modestly, and additional techniques will be needed to utilize the rest of the unused memory bandwidth.

Recognizing there is substantial locality in graph algorithms is one of our results that most strongly contradicts prior conventional wisdom. The implication is that graph algorithm implementations on current hardware platforms should optimize for cache locality. When designing a new architecture specialized for graph algorithms, it should be designed so it can exploit locality. Hardware managed caches are one of the most robust and easiest means to exploit locality in current processors, but perhaps something specialized for graph algorithms will work better. Ignoring locality and designing a system for peak random memory bandwidth (e.g. GUPS) unnecessarily increases the cost of the system.

Leveraging locality will decrease cost in a number of ways. By exploiting locality, less data needs to go off-chip for the same system throughput, so the system can reduce not only manufacturing costs by building less off-chip bandwidth, but it can also reduce energy consumption as data will move shorter distances on average.

Overall, we see no perfect solution to the performance challenges presented by graph algorithms. Many techniques can improve performance, but all of them will have quickly diminishing returns, so greatly improving performance will require a multifaceted approach.

Chapter 6

GAIL: Graph Algorithm Iron Law

In this chapter, we build on the insights of the workload characterization from the previous chapter to create the *Graph Algorithm Iron Law (GAIL)*. GAIL is a simple model that eases understanding of graph algorithm performance across abstraction layers.

6.1 Introduction

To support the growing interest in applications using graph algorithms, there has been a corresponding growth in graph-processing acceleration research. This interest in improving graph processing performance is not confined to any one area, and is ongoing at all layers of the computational stack (algorithms, implementation, frameworks, and hardware platforms). As each new innovation is shown to be beneficial by demonstrating a speedup, it is often unclear what causes the speedup since many of the recent innovations span multiple layers of the stack.

Execution time is the most important metric, but is unfortunately not instructive on its own. It allows us to quantify how much faster one execution is than another, but for those executions to be comparable using time, many parameters should be kept the same (e.g., input graph and graph kernel). Using a rate, such as traversed edges per second (TEPS), gives an idea of execution speed and the potential to compare executions using different input graphs. However, TEPS can be misleading, especially if the ways edges are counted for TEPS differ.

According to the Graph 500 [66] benchmark specifications, TEPS is the ratio of the number of undirected input edges (including duplicates and self-loops) in the traversed connected component and the execution time. Since this rate is often in the millions or billions, TEPS is often scaled to become Mega-TEPS (MTEPS) or Giga-TEPS (GTEPS). A common mistake, which is misleading, is to count an undirected edge as two directed edges. This mistake changes TEPS by a factor of two, which is often greater than the margin of improvement for a new optimization.

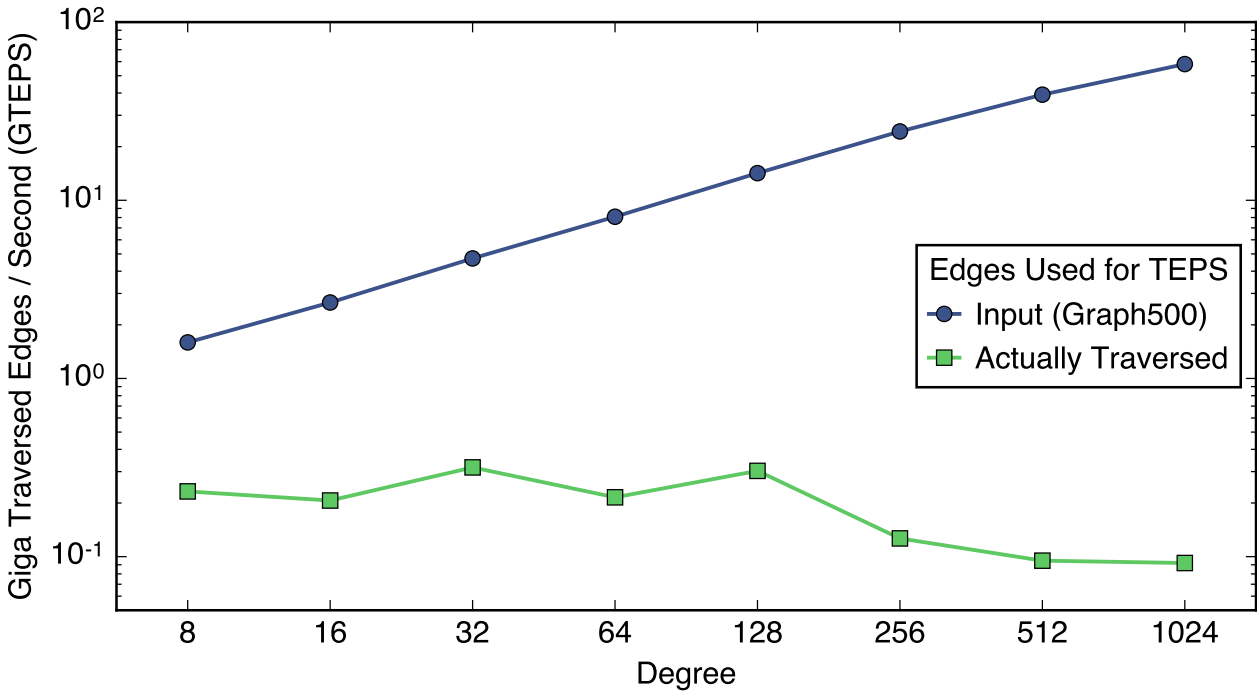

Figure 6.1: Impact of calculating TEPS based on input edges or actual edges traversed when scaling the degree of a 8 million vertex synthetically-generated Kronecker graph. Uses GAP direction-optimizing BFS implementation executing on IVB.

Our direction-optimizing BFS algorithm (Chapter 3) exemplifies how an optimized algorithm can further complicate the meaning of TEPS. The Graph500 definition of TEPS implicitly assumes that each input edge is traversed exactly once as done by the conventional top-down BFS approach, but our direction-optimizing BFS often skips a large fraction of the graph. If the TEPS rate is calculated based only on the number of actual edges traversed, a faster algorithm could appear to be slower due to a lower TEPS rate. Calculating TEPS based on the number of input edges does capture the speedup of an optimized algorithm, and it is similar in spirit to measuring GFLOPS for an optimized matrix multiplication algorithm such as Strassen in terms of the classic $O(N^3)$ algorithm. Unfortunately, calculating TEPS based on the number of input edges for an optimized algorithm has the drawback of making TEPS somewhat artificial in that the algorithm is only effectively – but not actually – traversing that many edges per second. With direction-optimizing BFS, increasing the degree of a graph can artificially inflate the GTEPS rate [95]. Figure 6.1 shows that for our direction-optimizing BFS implementation, GTEPS calculated with input edges scales with degree even though GTEPS calculated by actual edges traversed decays. This phenomenon is possible because increasing the degree increases the number of input edges at a rate greater than the number of edges our BFS algorithm actually needs to traverse.

The shortcomings of TEPS demonstrate that a single metric cannot be used, and instead it should be complemented by other metrics. For example, reporting TEPS would be less

confusing if the number of traversed edges was also reported. Furthermore, the number of traversed edges only considers algorithmic impacts and fails to consider architectural impacts. In this chapter, we introduce the *Graph Algorithm Iron Law (GAIL)* [17]. In addition to incorporating execution time and the number of traversed edges, GAIL also includes memory requests, since the memory system is often the biggest architectural bottleneck (as we observe in Chapter 5). Instead of simply quantifying which execution is faster as execution time and TEPS do, GAIL provides a high-level intuition as to the cause of performance differences. With GAIL, we concisely factor out the performance impacts from the algorithm, the implementation, and the hardware platform. In the rest of this work, we describe GAIL in detail and demonstrate its utility with case studies considering the impact of algorithm, implementation, input graph, and hardware platform.

6.2 Graph Algorithm Iron Law

Algorithms continue to be the most important factor for performance. Complexity analysis, the classic tool for evaluating algorithms, can sometimes be less instructive for high-performance graph-processing evaluations. Worst-case analysis often yields overly pessimistic performance bounds that differ greatly from the common case. Additionally, the amount of algorithmic work for many advanced graph algorithms depends heavily on the input graph topology, which is often difficult to quantify. Furthermore, as the field matures, many innovations will be implementation optimizations and not algorithmic optimizations, but these innovations would appear to provide no improvement if compared analytically. A good alternative in these challenging scenarios is to empirically measure the amount of algorithmic work.

Once the impact of the algorithm is understood, it is important to understand the potential and limitations of the hardware platform. Depending on the input graph and the algorithm, there can be a variety of hardware performance bottlenecks; however, graph algorithms are typically memory bound and not compute bound. As we demonstrate in our workload characterization in Chapter 5, the memory bottleneck is due to memory latency much more often than it is due to memory bandwidth, because memory bandwidth is typically underutilized. A modest cache hit rate can cause cache misses to be so rare that an insufficient number of them fit into the instruction window to fill all of the outstanding memory request slots. With an insufficient number of outstanding memory requests, the processor is unable to fully utilize the memory bandwidth. The challenge of fitting a sufficient number of cache missing instructions into the instruction window leads to a common tradeoff: improving cache hit rates results in memory bandwidth becoming further underutilized. Other factors (branch mispredictions, low instruction-level parallelism, and temporal variation) can also hinder memory bandwidth, but good locality is often the most impactful. Given all of this, the most important architectural features affecting graph processing execution are cache effectiveness and memory bandwidth utilization.

Taking the insights we have learned above, we present a simple model to understand graph algorithm performance. Analogous to how the Iron Law of CPU performance [57, 129] factors execution time into the product of the number of instructions executed, cycles per instruction, and the cycle time, our Graph Algorithm Iron Law (GAIL) factors execution time into the product of the number of traversed edges, the number of memory requests per traversed edge, and the inverse of the memory bandwidth:

$$\frac{\text{execution time}}{\text{kernel}} = \frac{\text{traversed edges}}{\text{kernel}} \times \frac{\text{memory requests}}{\text{traversed edge}} \times \frac{\text{seconds}}{\text{memory request}}$$

This model combines the three characteristics we find most relevant for graph algorithm performance: algorithmic efficiency, cache effectiveness, and memory bandwidth utilization. Naturally, since GAIL breaks execution time into the product of three metrics, lower is better for all metrics. To measure the amount of algorithmic work, we use edges examined instead of vertices examined because the amount of work per edge is roughly constant, while the amount of work per vertex can vary dramatically (it is often a function of its degree). With the help of execution time and edges examined, we break down communication (DRAM memory requests) into two instructive rates that measure cache effectiveness (second term) and memory bandwidth utilization (third term). The third term (seconds per memory request) is best thought of as inverse of bandwidth rather than the average time per memory request. If there is more than one memory request outstanding, this metric will understandably become smaller than the actual average memory latency. For memory requests, we are referring to cache misses that go to DRAM. We do not count cache hits since we assume out-of-order execution can hide their latency.

Although GAIL may appear simple, there are some noteworthy elements hidden inside its formulation. The product of the first two terms is the number of memory requests per kernel, which is proportional to the amount of data moved from DRAM. The product of the second term and the third term is seconds per traversed edge, which is the inverse of TEPS.

When counting memory requests, GAIL users should include prefetches in addition to cache misses. Although counting prefetches can be more arduous than counting only cache misses, including prefetch traffic is worthwhile because it increases the accuracy of GAIL. When the prefetches are beneficial, accounting for that traffic prevents underreporting the number of memory requests necessary to complete the kernel. When the prefetches are not helpful, counting prefetches overstates the number of memory requests necessary for the kernel, but we consider potentially overstating to be less misleading than not counting prefetches and underreporting. Furthermore, most modern processors are designed to decrease the amount of hardware prefetching to keep these unneeded prefetches from being too detrimental to performance. Unused prefetches will consume memory bandwidth, but a reasonably designed system will give cache misses priority over the hardware prefetcher. Throughout our experiments on IVB, we never observed the hardware prefetcher consuming all of the remaining memory bandwidth.

The GAIL metrics provide intuition by decomposing performance into the amount of work to do (traversed edges), work efficiency (memory requests per traversed edge), and

Graph	Description	Vertices (M)	Edges (M)	Degree	Degree Distribution	Diameter
Road	USA road network [50]	23.9	58.3	2.4	bounded	6,277
Kron	Kronecker generator [66, 94]	134.2	2,111.6	15.7	power	5
Urand	uniform random [58]	134.2	2,147.4	16.0	normal	6

Table 6.1: Graphs used for case studies

throughput (inverse bandwidth). The coarse-grained intuition GAIL provides can be instructive to a variety of potential users. Algorithm designers can use it to demonstrate the work reduction (fewer edges examined), and doing this empirically can be helpful when it is difficult to quantify analytically. A framework developer can demonstrate the efficiency of their implementation by achieving either higher memory bandwidth utilization or fewer memory requests per traversed edge. Even a hardware platform designer could use GAIL to demonstrate an improvement in memory bandwidth utilization.

6.3 Case Studies Using GAIL

For any evaluation, introspective measurements should be taken to attempt to explain the results, and some of the metrics used may be specific to that study. GAIL is useful as a high-level starting point, as it is a simple way to verify if the improvements reduce the amount of algorithmic work, increase locality, or improve memory bandwidth utilization. To demonstrate the utility of GAIL, we perform case studies to investigate tradeoffs in software implementations and changes to the hardware platform.

For our case studies, we use the GAP Benchmark reference implementations (Chapter 4). We select three input graphs for size and topological diversity (Table 6.1). A proper evaluation will use more input graphs, but we believe these brief case studies are sufficient to demonstrate GAIL's utility. For more information on the graphs and their relevant topological properties, please refer to Section 2.3.

Table 6.2 specifies the IVB and T4 platforms we use to perform our case studies demonstrating the utility of GAIL. We use IVB for all of our case studies in this section and IVB is the same platform we use in the rest of this work. For our final case study, we also use T4 to provide a different hardware platform for comparisons to IVB.

Understanding Software Implementation Performance

In our first case study, we use GAIL to examine the impacts of algorithmic innovations and implementation optimizations on BFS traversing the kron graph. We utilize the three

	IVB	T4
Architecture	Ivy Bridge	S3 Core
Model	Intel E5-2667 v2	SPARC T4-4
Released	2013	2011
Clock rate	3.3 GHz	3.0 GHz
# Sockets	2	4
Cores/socket	8	8
Threads/core	2	8
LLC/socket	25 MB	4 MB
DRAM Capacity	128 GB	1024 GB
Maximum Achieved Memory Request Rate	1204 M/s (0.831 ns/req)	2049 M/s (0.488 ns/req)

Table 6.2: IVB and T4 systems used for case studies

Implementation	Time (s)	Memory Requests (M)	Traversed Edges (M)	Memory Requests / Edge	ns / Memory Request
Top-down	3.964	3,292.88	4,223.22	0.780	1.204
Top-down-bitmap	2.255	1,024.31	4,223.22	0.243	2.201
Direction-optimizing	0.424	209.50	183.78	1.140	2.022

Table 6.3: GAIL metrics for BFS implementations traversing kron using IVB

BFS implementations introduced in Section 3.5: top-down, top-down-bitmap, and direction-optimizing. For these implementations traversing kron, Table 6.3 shows the GAIL metrics (last three columns) as well as the raw data that produces them (second, third, and fourth columns).

As a simple example, we first examine the GAIL metrics for the top-down baseline in Table 6.3. As expected, the number of traversed edges is twice the number of input edges (Table 6.1) since each undirected input edge is traversed once in each direction. The cache provides some benefit, as the number of memory requests per traversed edge is less than one. From the last GAIL term (1.204 ns / memory request) we can see that the top-down execution uses 69% of the platform's memory bandwidth when compared to the maximum we achieve with our bandwidth microbenchmark (Section 5.3).

The bitmap optimization in top-down-bitmap does improve performance relative to top-down (1.75× speedup to 2.255 seconds), but from the GAIL metrics we can clearly see the optimization is not algorithmic since it traverses the same number of edges. The benefit of the bitmap optimization is visible in the reduction in memory requests per edge, as fewer edge traversals miss in the LLC because the cache-resident bitmap is able to satisfy many

of the edge traversals. In spite of utilizing less memory bandwidth (higher ns per memory request), top-down-bitmap is still faster than top-down because of the larger reduction in memory requests per traversed edge. The GAIL metrics easily decompose this beneficial tradeoff of one metric (cache locality) improving by more than another metric (memory bandwidth utilization) worsens.

From the execution times (Table 6.3), we see the direction-optimizing implementation is the fastest, but without more information we cannot be sure of the cause. From our experience in Chapter 3, we have intuition that the direction-optimizing implementation derives its speedup by performing algorithmically less work, but in Figure 3.15 the speedups for the direction-optimizing implementation are substantially less than the reductions in algorithmic work. The same phenomenon occurs in this case study, as the direction-optimizing implementation's 23× reduction in traversed edges is substantially greater than its 5.3× speedup over the top-down-bitmap implementation. Now, unlike in Chapter 3 in which we only use algorithmic tools, with the help of the GAIL metrics we can confirm that the direction-optimizing implementation traverses edges at a slower rate because it misses the LLC more often and it utilizes less memory bandwidth. The higher cache miss rate is probably due in large part to bottom-up traversals terminating early once a parent is found. The early terminations will reduce spatial locality, causing fewer words per transfered cache line to be used, which in turn increases number of memory requests per traversed edge. The memory bandwidth utilization is reduced because the more serial nature of the inner-loop of the bottom-up approach reduces the amount of application MLP.

Understanding Algorithmic Performance Impacts

GAIL is also useful to understand tradeoffs between different layers of the computational stack. Changing an algorithm can effect not only the amount of algorithmic work to do, but also how well the implementation executes on a given hardware platform. As a case study to investigate these tradeoffs, we consider the delta-stepping algorithm and vary its Δ parameter. Delta-stepping is an optimized algorithm for single-source shortest paths (SSSP) that attempts to pragmatically increase parallelism without adding substantial algorithmic overhead [107]. Delta-stepping can be thought of as a generalization of the classic Dijkstra [49] and Bellman-Ford algorithms [172].

Dijkstra's algorithm – the classic optimal algorithm for SSSP – often lacks sufficient parallelism to fully utilize current multicore platforms. Dijkstra's algorithm obtains its algorithmic efficiency by processing vertices in order of increasing distance from the root vertex, and this order guarantees each vertex is processed only once. Unless multiple vertices are the same distance from the root vertex, only one vertex can be processed at a time by Dijkstra's algorithm. To schedule the vertex processing order, Dijkstra's algorithm uses a priority queue of unvisited vertices sorted by distance, and updating the priority queue can often become another bottleneck restricting parallelism.

The delta-stepping algorithm increases parallelism by relaxing the strictness of the vertex processing order. Instead of perfectly sorting all vertices by distance, delta-stepping coarsely

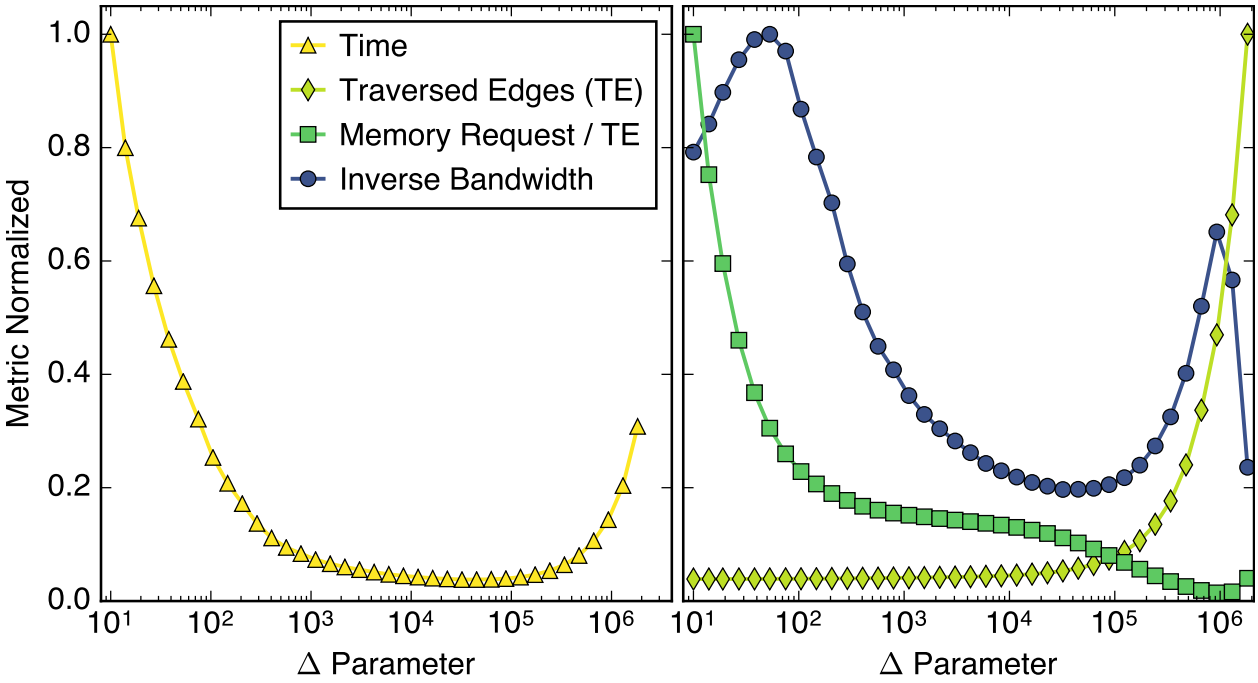

Figure 6.2: GAIL metrics for delta-stepping implementation while varying Δ-parameter traversing the road graph using 8-cores on the IVB platform. From the GAIL metrics, we see the U-shape in execution time is caused by the L-shape of the number of memory requests per traversed edge and the backwards L-shape for the number of traversed edges.

radix sorts the vertices by distance into buckets of width Δ. Delta-stepping allows all vertices in the current minimum distance bucket to be processed in parallel. By processing vertices in parallel, there is the potential some vertices will be processed prematurely and thus need to be processed again later, but in practice, this happens rarely if the Δ parameter is sufficiently small. The width of the buckets is parameterized by Δ, which is what gives the algorithm its name. Changing the Δ parameter to change the width of the buckets is a tradeoff between exposing more parallelism and performing more redundant algorithmic work. Delta-stepping generalizes the classic SSSP algorithms. If $\Delta = 1$ (or whatever the minimum distance increment is), delta-stepping behaves like Dijkstra's algorithm. If $\Delta = \infty$, all vertices are in the same bucket and delta-stepping behaves like the Bellman-Ford algorithm.

In Figure 6.2, we sweep the Δ parameter while searching the road graph. We use the GAP Benchmark's reference implementation of delta-stepping (Chapter 4), and we execute on a single socket of the IVB platform. As we vary Δ, the execution times forms a U-shape (left side of Figure 6.2), indicating the optimal range for Δ is somewhere in the middle and the optimal range is quite large (spans an order of magnitude). From our algorithmic intuition, we expect a Δ that is too small to limit "parallelism" and a delta too large to be burdened by redundant work, but with GAIL we can fully understand the tradeoffs involved with changing Δ (right side of Figure 6.2). First, we examine the number of edges traversed, and we see

that as expected, increasing Δ does increase the number of edges traversed. It is interesting to note that the number of edges traversed grows rather slowly as Δ increases, indicating there is plenty of slack available in the vertex processing order without causing substantial redundant work. Second, we observe the number of memory requests per traversed edge decreases substantially as we increase Δ. With a larger Δ, more vertices are processed per super-step, which allows for better amortization of compulsory memory traffic. The inverse bandwidth generally has the same U-shape as the execution time with some second-order effects at the extremes. For very small Δ, there is so much extra data to move and it is so predictable that bandwidth improves (more benefit from prefetching). Additionally for very large Δ, the majority of vertices can be processed simultaneously, which improves spatial locality and once again allows the prefetcher to help more.

From a purely algorithmic point-of-view without GAIL, we would attribute performance increases from increasing Δ to increased "parallelism" (more cores busy), but from GAIL we can see that this increased algorithmic parallelism actually helps by improving communication efficiency. From GAIL, we can see that the U-shape in execution time from a Δ-sweep is the combination of the L-shape for the number of memory requests per traversed edge and the backwards L-shape for the number of traversed edges.

Understanding Parallel Scaling Performance

To demonstrate the utility of GAIL for evaluating hardware platforms, we vary the number of cores our workload uses. By varying the number of cores we allocate, we are not only increasing the potential computational throughput, but we are also increasing the potential for cache thrashing in the LLC and increasing synchronization overheads due to an increased number of participants. For these strong scaling experiments, we use the direction-optimizing BFS implementation executing on the IVB platform. Figure 6.3 shows the GAIL metrics for two graphs: kron and road. The number of traversed edges is not shown because the implementation is deterministic and so the number of traversed edges does not change for a given graph.

Strong scaling on the kron graph (left side of Figure 6.3) is an example of successful parallel scaling, as the implementation achieves a 10× speedup when using 16 cores. The speedup is driven by the system successfully utilizing more memory bandwidth, as the reduction in execution time closely tracks the inverse memory bandwidth (ns per memory request). The increase in memory bandwidth utilization is indicative of more parallelism in the computation (more cores). Crucial to the success of this scaling is that the amount of communication (memory requests / traversed edge) does not change significantly.

Strong scaling on the road graph (right side of Figure 6.3) is an example of lackluster parallel scaling. The road graph's high diameter and small size complicate parallel scalability for our implementation. Since it is a high diameter graph, the direction-optimizing implementation will never switch into the bottom-up approach and will perform the entire search top-down. Since the graph road has fewer edges and a high diameter, there is less work per step and more steps (greater synchronization overhead), further complicating par-

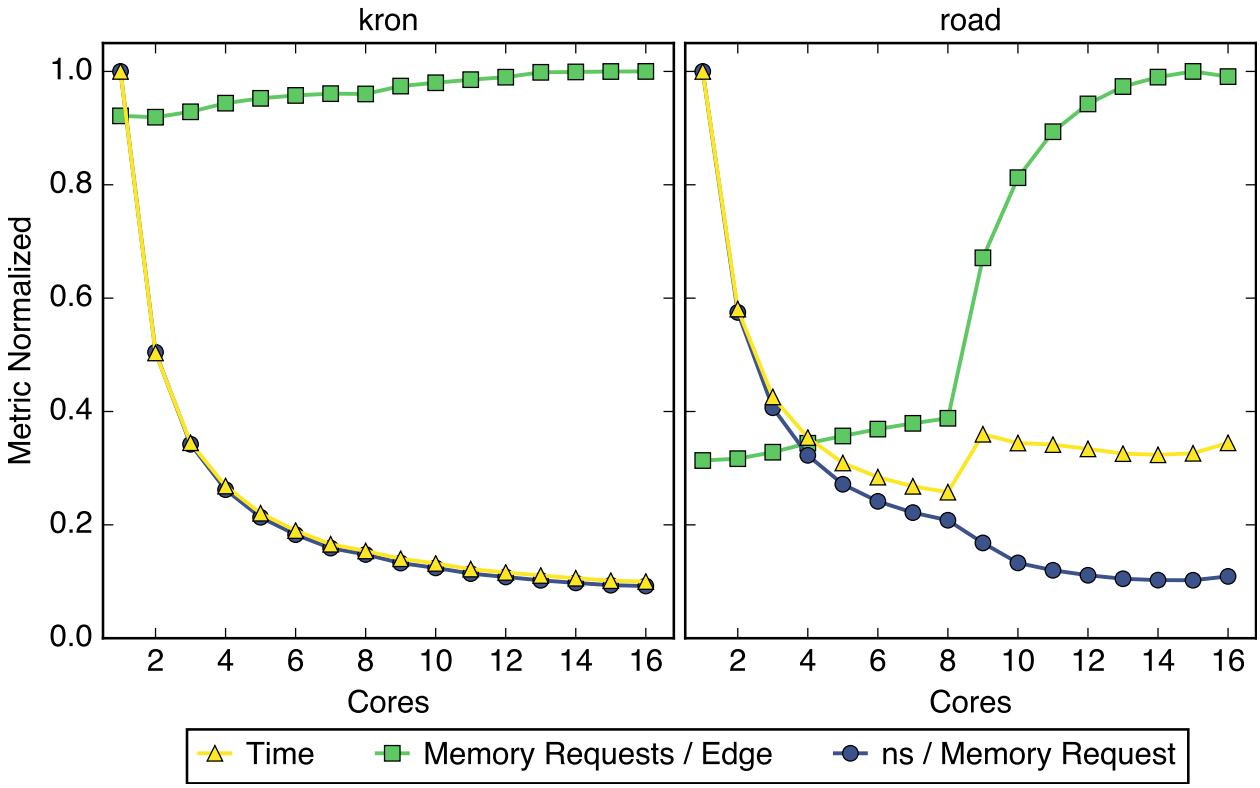

Figure 6.3: GAIL metrics for strong scaling (varying number of cores utilized) on IVB for direction-optimizing BFS implementation traversing the kron graph (left) and the road graph (right). Since the implementation is deterministic, the traversed edges GAIL metric is constant for each graph (not shown).

allel scalability. Using only one socket (up to 8 cores), achieves a 4× speedup by utilizing more bandwidth than it moves additional data. Once the implementation starts using both sockets to traverse the road graph, it actually goes slower (hitch at 9 cores). With the help of GAIL, we can clearly see the cause is a sharp increase in the number of memory requests per edge that overpowers the increase in utilized memory bandwidth. Since the increase in memory bandwidth utilization is somewhat comparable to the successful kron parallel scaling, this is an indication to the hardware designer that the cache is likely thrashing. To the software implementor, this scaling behavior suggests that the graph may be too small to continue strong scaling on this 16-core platform.

Understanding Hardware Platform Performance Impacts

In the this final case study, we use GAIL to compare the IVB and T4 hardware platforms from Table 6.2. Even when the performance of the two hardware platforms is comparable, they may achieve their performance through different means. By using GAIL for this comparison,

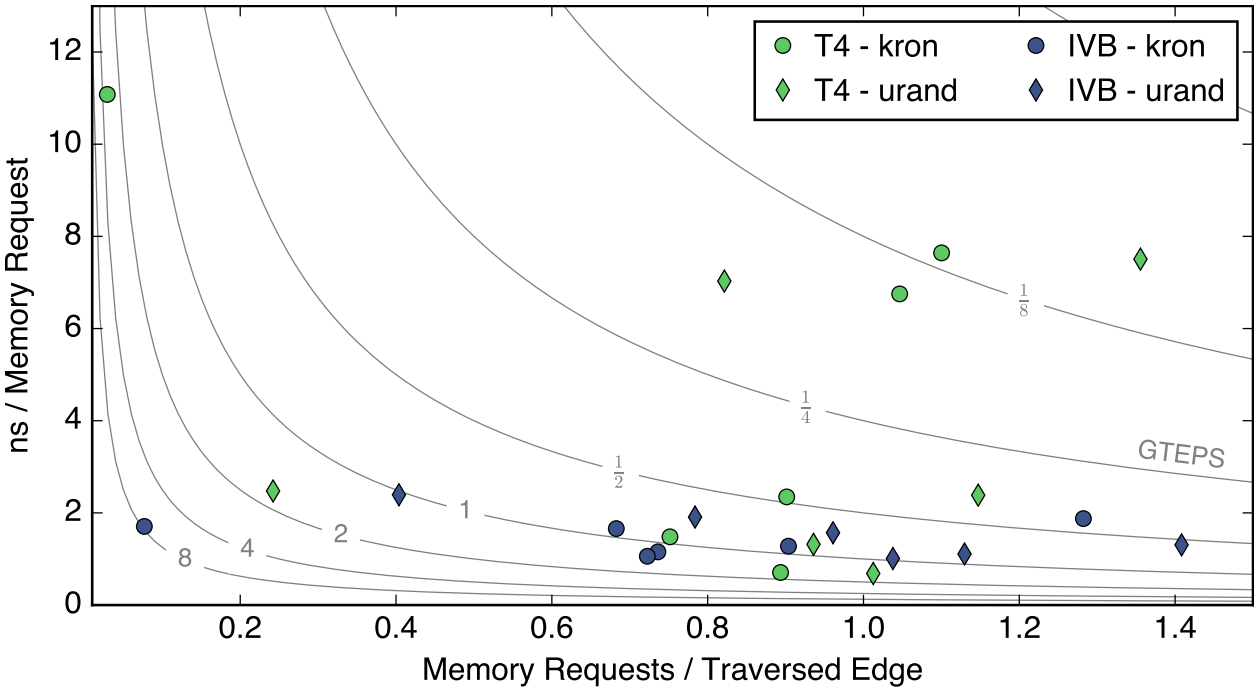

Figure 6.4: GAIL metrics (memory requests per traversed edge versus inverse memory bandwidth) for GAP Benchmark kernels executing on IVB and T4 to process the kron and urand graphs. Contours show GTEPS for edges actually traversed.

we can understand how each platform achieves its performance. For this analysis, we execute the GAP Benchmark reference code on each platform to process the kron and urand graphs. IVB is representative of a "scale-out" server in a datacenter while T4 is representative of a standalone "scale-up" server. Although T4 is a few years older than IVB, it is a substantially larger system capable of nearly double the peak theoretical throughput. T4's cores are weaker than IVB's cores, but they support a greater number of active threads. IVB has the advantage of substantially greater LLC capacity. Comparing these platforms indirectly evaluates tradeoffs between more small cores versus fewer large cores and more memory bandwidth versus more on-chip memory.

Before comparing the two hardware platforms head-to-head, we first calibrate our expectations for the second and third GAIL metrics in Figure 6.4. We do not visualize the first metric (traversed edges) since we use the same implementation on both platforms so the number of traversed edges is either the same (BC, BFS, PR, and TC) or close to the same (CC and SSSP). In Figure 6.4, the contours represent traversed edges per second so moving towards the origin represents higher throughput achieved by either better memory bandwidth utilization (moving down) or reduction in memory communication (moving left). From Figure 6.4, we see both platforms have a range of memory bandwidth utilizations and cache efficiencies depending on the input graph and kernel. The ranges for the two platforms largely overlap, indicating the workload differences are more impactful than the hardware

Graph	BC	BFS	CC	PR	SSSP	TC
kron	0.983	3.411	2.499	0.826	5.230	2.481
urand	0.818	3.513	2.186	0.662	5.385	0.620

Table 6.4: Speedups executing GAP benchmark kernels of IVB relative to T4

differences. Comparing the two platforms, there are some cases in which T4 utilizes substantially less bandwidth than IVB (points corresponding to BFS and SSSP). Although harder to see, there are also some points in which T4 utilizes more bandwidth than IVB. Comparing the two input graphs for a given kernel and platform, the points for kron are typically to the left of the points for urand, demonstrating the locality benefits of kron's power-law degree distribution in practice.

Table 6.4 shows the speedups of IVB relative to T4. We see that although IVB is substantially faster in 7/12 cases, in the remaining 5/12 cases T4 is faster. The kernel implementations (BC and PR) that execute faster on T4 are the more regular implementations, and this suggests T4's slowdowns relative to IVB may be due to poor parallel utilization. The more irregular implementations are more difficult to balance across threads, and T4 must balance across 8× as many threads. This behavior is best exemplified by the TC implementation where T4 is faster than IVB on urand but slower on kron. For our TC implementation, processing the urand graph results in work units more regular than the those for the kron graph. Our TC implementation recognizes kron is scale-free, and this observation triggers an optimization to relabel the graph by degree, which increases the variance of the work units. On IVB, OpenMP's dynamic scheduling is able to successfully adaptively balance these varied work units, but when scaled to 8× as many threads, dynamic scheduling on T4 is not as successful. This result demonstrates the challenge of obtaining great parallel utilization for optimized algorithms which are often more irregular.

Figure 6.5 shows IVB's improvement over T4 on the second and third GAIL metrics relative to speedup. On the left side of Figure 6.5, we observe no correlation between differences in memory requests per traversed edge and speedup. Despite a sizable LLC capacity advantage, IVB does not obtain a substantial reduction in memory requests per traversed edge. The lack of substantial difference in memory requests per traversed edge also suggests that IVB and T4 are caching the same working set, and IVB's LLC will need to be substantially larger to contain the next largest working set.

On the right side of Figure 6.5, we see a strong correlation between speedup and the improvement of inverse bandwidth. This indicates that for this platform comparison, higher overall performance tracks memory bandwidth utilization because the number of traversed edges and the number of memory requests per traversed edge do not vary substantially across platforms. The outlier is TC processing kron, and this point is skewed by IVB's prefetcher aggressively using available memory bandwidth, which artificially increases the number of memory requests. Overall, this case study demonstrates the importance of not only peak theoretical throughput, but the importance of being able to utilize that throughput.

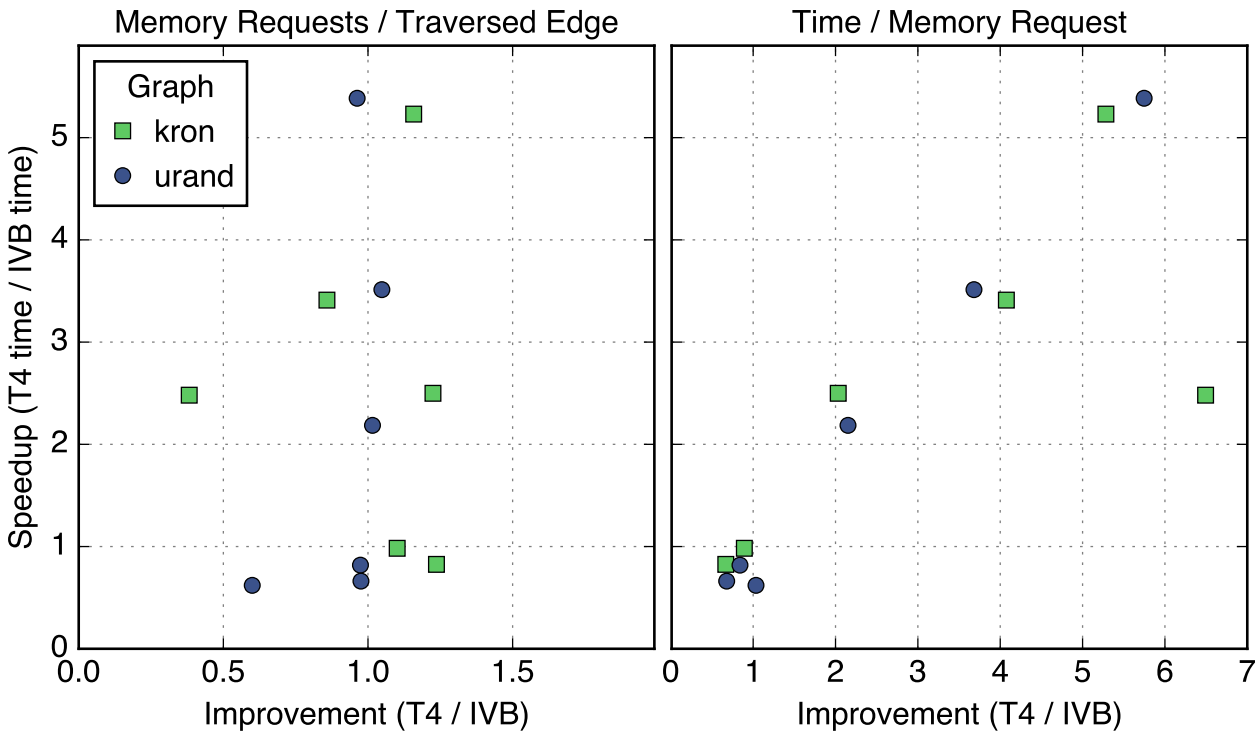

Figure 6.5: Speedup of IVB over T4 versus IVB's improvement over T4 in GAIL metrics (memory requests per traversed edge versus inverse memory bandwidth) for GAP benchmark kernels processing the kron and urand graphs.

6.4 Using GAIL to Guide Development

In the previous section, we demonstrate GAIL's utility with experimental case studies to understand existing performance differences, and in this section, we discuss how GAIL can also be used to guide algorithm, software, or hardware development. At the start of a new project, a designer can pick which GAIL metrics they hope to improve, and which metrics they hope to hold constant or degrade only slightly. With GAIL, simple back-of-the-envelope math can provide bounds on potential performance improvements when considering potential bottlenecks. Before even prototyping an idea, these simple calculations can help the designer decide if an idea has sufficient potential. As the project progresses, the GAIL metrics can give the designer further guidance on which challenges are the most important to solve. In this section, we use GAIL to discuss probable tradeoffs experienced when creating a new algorithm, a new software framework, or a new hardware platform.

To create a new optimized algorithm, its designer will attempt to reduce the number of traversed edges, but to be faster in practice, the new algorithm must not worsen the other GAIL metrics by more than the algorithmic improvement. For example, the new algorithm could be slowed if it has a more irregular memory access pattern that increases the number of memory requests per traversed edge. The new algorithm could also be more difficult to

parallelize, and a reduction in throughput could be visible using GAIL as an increase in inverse bandwidth. Our direction-optimizing BFS algorithm exemplifies these challenges, as its increased irregularity worsens cache locality and bandwidth utilization, but due to a drastic reduction in traversed edges, it provides an overall speedup.

The challenges of creating a new high-performance graph processing framework can also be analyzed with GAIL. Presumably, the new framework will provide a programming model to increase its users' productivity, but it is important that the programming model does not preclude optimized algorithms. Being forced to use less optimized algorithms could increase the number of edges traversed and dampen any speedups. Additionally, the framework needs to lay out its data structures compactly and intelligently to not increase the number of memory requests per traversed edge. Finally, the framework needs to allow for high throughput. This challenge is more than good parallel scalability, as in practice, the framework may be bottlenecked by poor memory bandwidth utilization. If the framework adds too many instructions, the increase in IPM can decrease memory bandwidth utilization (as we observe in Chapter 5).

GPUs provide a concrete example of a substantially different hardware platform, and the challenge of obtaining peak graph processing performance with GPUs are similar to the challenges a graph processing hardware accelerator will face. GPUs have the potential for substantial throughput, as they can execute many threads simultaneously and have tremendous memory bandwidth. Unfortunately, this throughput potential comes at the price of reduced programmability and less on-chip memory. Prior work on high-performance graph processing on GPUs experiences these challenges, but their penalties can be better explained with GAIL by examining the number of edges traversed and the number of memory requests per traversed edge. Some algorithmic optimizations may be impractical for GPUs, as GPUs often require high control convergence in order to obtain high utilization. Akin to the challenge faced by software frameworks of providing a sufficiently flexible programming model, being forced to use a less optimized algorithm could increase the number of traversed edges. With the reduced on-chip memory available on GPUs, the implementation also needs to be careful not to increase the number of memory requests per traversed edge.

6.5 Frequently Asked Questions

In talking to others about using GAIL, some questions have arisen:

Q: What if GAIL users have different definitions for what constitutes a traversed edge?

A: Different traversed edge definitions will of course change the numerical results of GAIL, but by whatever factor the number of traversed edges is scaled will also inversely scale the number of memory requests per traversed edge. For example, deliberately undercounting traversed edges to appear algorithmically better will result in appearing to perform more memory requests per edge (worse). This concern should generally not be problematic since

what constitutes a traversed edge is typically unambiguous and most evaluations will be performed by the same evaluator.

Q: Are the inputs to GAIL (time, traversed edges, and memory requests) the only important metrics for graph algorithm performance?

A: No, but we believe those to be the three most important. Branch mispredictions can definitely hinder performance [68]. Considering the number of instructions per traversed edge is another interesting metric for implementation efficiency, but we find it to be less instructive than memory requests per edge. Instruction throughput for graph algorithm execution is much more likely to be stalled by waiting on memory requests than a lack of available function units to execute ready instructions [19].

Q: Could GAIL be modified to consider energy?

A: Yes. Replacing time with energy in the GAIL equation results in an interesting new term: joules per memory request. Unfortunately, this modification to the GAIL equation may be less instructive due to tradeoffs between dynamic power and static power [41].

Q: Does GAIL work for distributed memory or semi-external memory implementations?

A: GAIL is designed for single-node shared memory systems since its metrics focus on the platform's most impactful architectural features (cache utility and memory bandwidth). For other platforms, substituting memory requests within GAIL for the other platform's most important bottleneck could make a similarly instructive equation. For example, for distributed memory (cluster) implementations [31], counting network packets will be more helpful. For semi-external memory (SSD/hard drive) implementations [92], counting blocks read from storage could be more useful.

Q: Does GAIL work for non-traditional architectures (those without caches)?

A: Yes. The Cray XMT is such an architecture, with a high thread count and no processor caches [154]. Even without the cache acting as a filter, the number of memory requests and the rate they execute can vary substantially. For example, different algorithms could result in different numbers of memory requests per traversed edge. Different implementations or compiler optimizations could change whether variables are kept in architectural registers or re-read from memory.

Q: Does GAIL work for all algorithms?

A: GAIL is designed for graph algorithms and it only requires a notion of traversing an edge. For algorithms that do not operate on graphs, we believe other analogous iron laws could be defined.

6.6 Conclusion

More specialized metrics should be a part of many graph processing evaluations, but GAIL provides a simple starting point by factoring out the tradeoffs between algorithmic work (traversed edges), cache locality (memory requests per traversed edge), and memory bandwidth utilization. GAIL is easy to use, since beyond execution time, the only additional data that

needs to be collected is the number of memory requests and the number of traversed edges. If widely adopted, GAIL could allow for concise comparisons with metrics already familiar to the community. Fundamentally, GAIL is encouraging the use of deeper and more instructive metrics. A new research contribution is much more useful if the community understands why the contribution is faster, rather than simply knowing it outperforms the predecessor.

Chapter 7

Propagation Blocking

In this chapter, we introduce *propagation blocking*, a graph algorithm implementation optimization to improve spatial locality in order to reduce memory communication. In Chapter 5, we observe that many graph processing workloads have locality and are consequently not memory bandwidth-bound, but there are a handful of low-locality workloads that break this trend. Our propagation optimization targets these low-locality graph workload executions. Propagation blocking is most advantageous when the input graph is sparse and has a large number of vertices relative to the cache size of the execution platform. We evaluate the effectiveness of propagation blocking on PageRank and demonstrate memory communication reductions on real hardware.

7.1 Introduction

The bounty of transistors provided by Moore's Law has enabled increased computational speed and throughput, but total communication bandwidth has failed to keep up. As a consequence, some low arithmetic intensity workloads are often bottlenecked by communication on today's platforms. For these communication-bound workloads, the only way to improve performance is to either increase their effective memory bandwidth or decrease the amount of communication. Reducing communication can also save energy, as moving data consumes more energy than the arithmetic operations that manipulate it [41].

The amount of memory communication needed to execute a graph processing workload is a function of many things including: the cache size, the graph size, the graph layout, and the software implementation. To reduce communication, prior work has examined improving the graph layout or reordering the computation to increase locality, and these optimizations are often beneficial. However, there are some graphs that are less amenable to layout or reordering transformations. Low-diameter graphs, such as social networks, are often such stubborn graphs with low locality.

In this chapter, we present *propagation blocking*, an optimization that improves spatial locality of low-locality graph workloads. By improving the spatial locality of a workload, our

optimization accelerates the workload by reducing the amount of memory communication. Performing propagation blocking adds additional computation, but for a communication-bound workload, the benefit from improving spatial locality makes this tradeoff beneficial.

We select PageRank to evaluate propagation blocking, since it is often communication-bound due to its low arithmetic intensity. In Figure 5.8, the datapoints in the upper left corner that experience the lowest locality and use the most memory bandwidth correspond to PageRank processing the largest graphs in the study. In Chapter 5, we observe the tremendous impact the input graph's topology can have on the characteristics of a graph processing workload, and PageRank can be the most affected by a low-locality input graph. PageRank is an application of the linear algebra Sparse Matrix Multiplying Dense Vector (SpMV) kernel, so our propagation-blocking technique can also be applied to SpMV.

We evaluate our approach on a suite of eight real-world and synthetic graphs using hardware performance counter measurements. Compared to our baseline implementation, our new approach reduces communication by 1.6 – 3.6× on the seven low-locality graphs, but increases communication by 1.6× on the one high-locality graph. Compared to conventional cache blocking, our approach reduces communication by 1.0 – 2.5× on five of the low-locality graphs. It communicates 1.1-1.2× more than conventional cache blocking on the remaining two low-locality graphs that have fewer vertices and more edges per vertex (denser). Determining whether to use cache blocking or our proposed propagation blocking could be done at runtime based on the number of vertices and the degree.

7.2 Background on PageRank

PageRank [127] has emerged as a popular graph benchmark as it exposes many of the challenges of graph processing while still being simple enough to ease analysis and implementation (Chapter 4). PageRank determines the "popularity" of a vertex by summing the scaled popularities of the vertices that point to it. This analysis often results in cyclic dependencies, so PageRank typically iterates until the scores converge on a fixed point. PageRank scores can also be computed or approximated by other techniques including spectral methods, but in this work we focus on the power method.

The PageRank score for a vertex u is ($d = 0.85$):

$$PR(u) = \frac{1-d}{|V|} + d \sum_{v \in N^-(u)} \frac{PR(v)}{|N^+(v)|}$$

At the core of this computation is the propagation of a vertex's score $PR(v)$ scaled by its out degree $|N^+(v)|$ to the vertex u it points to. In other words, a vertex's score $PR(u)$ is the sum of the contributions (scaled scores) from its incoming neighbors $N^-(u)$. In our discussion of implementing PageRank, we focus on the propagation of scaled scores between vertices as the other multiplications or additions are on scalar or on often reused values.

We perform much of this work's analysis of PageRank from the graph algorithm perspective, but to ease some explanations, we sometimes take the sparse linear algebra perspective,

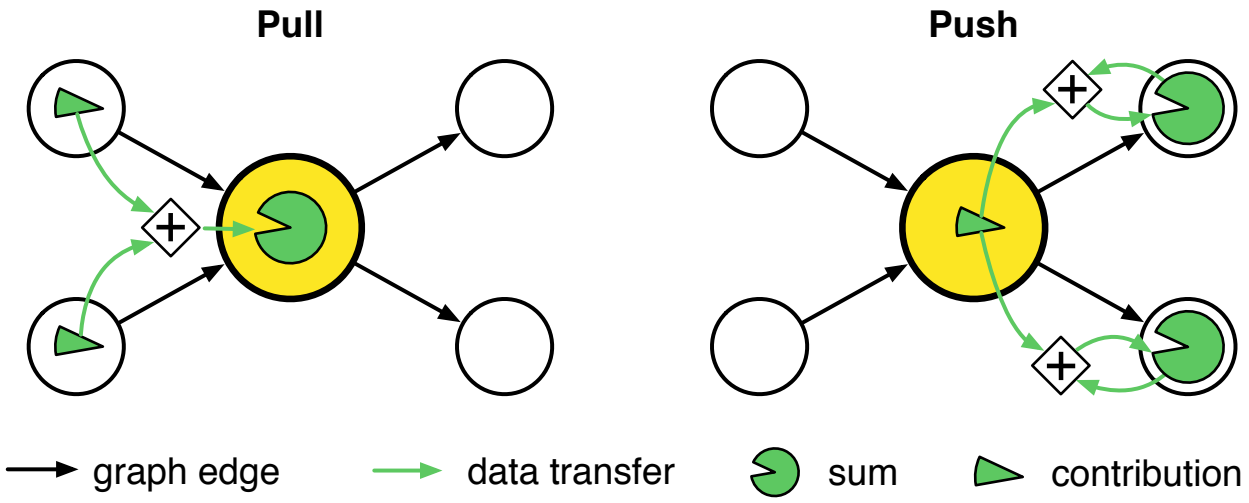

Figure 7.1: PageRank communication for pull direction (left) and push direction (right). In the pull direction, the active vertex (shaded) reads the contributions of its incoming neighbors and adds them to its own sum. In the push direction, the active vertex adds its contribution to the sums of its outgoing neighbors.

where the input graph can be viewed as a sparse adjacency matrix A such that each non-zero element A_{ij} represents an edge from vertex i to vertex j. The propagation and reduction of contributions in PageRank is an application of SpMV. There are two restrictions on SpMV within PageRank not present in generalized SpMV. First, the matrix A must be square since the rows and columns represent the same set of vertices. Second, the matrix is binary (graph is unweighted), so the indices themselves are the useful information as there are no associated values to read with them. Additionally, the binary matrix results in each output element being a sum reduction instead of a dot product. Our proposed technique does not rely on these restrictions, and it could be extended to handle generalized SpMV.

There are two directions to perform the computation (Figure 7.1): pull (akin to row-major SpMV) and push (akin to column-major SpMV). In the pull direction, each vertex reads the contributions of its incoming neighbors and computes its own score (Figure 7.2a). In the push direction, each vertex adds its contribution to the sums of its outgoing neighbors (Figure 7.2b). After propagating the contributions in the push direction, a later pass will use those sums to compute the new scores. The pull direction is often more efficient since it only reads each neighbors' contributions rather than doing an atomic add to each neighbor's sum. Implementing the pull direction requires the transpose graph if the graph is directed, as the computation will need to know the incoming edges for each vertex. Both directions compute the same result, and the distinction is whether the sum (pull) or the contribution (push) is the vertex value immediately reused. The tradeoffs between push and pull are analogous to the tradeoffs between performing breadth-first search top-down and bottom-up in Chapter 3.

```
scores[:] ← 1 / |V|
base ← (1 - d) / |V|
for i until max_iters do
  for u ∈ V do
    contributions[u] ← scores[u] / |N+(u)|
  end for
  for u ∈ V do
    sum ← 0
    for v ∈ N−(u) do
      sum ← sum + contributions[v]
    end for
    scores[u] ← base + d × sum
  end for
end for
```

(a) Pull direction

```
scores[:] ← 1 / |V|
sums[:] ← 0
base ← (1 - d) / |V|
for i until max_iters do
  for u ∈ V do
    contribution ← scores[u] / |N+(u)|
    for v ∈ N+(u) do
      sums[v] ← sums[v] + contribution
    end for
  end for
  for u ∈ V do
    scores[u] ← base + d × sums[u]
    sums[u] ← 0
  end for
end for
```

(b) Push direction

Figure 7.2: PageRank implemented in both directions

7.3 Locality Challenges for PageRank

Locality is extremely important to PageRank performance, as improved locality can reduce communication which in turn will improve performance. Although a given algorithm may have a predictable number of reads and writes based on its input size, those reads and writes may translate into a variable number of memory requests to DRAM based on cache locality. Although modern processors have multiple levels of caches, in this work we consider only a single cache level representative of the last-level cache (LLC). This is reasonable, as the bandwidth between the LLC and DRAM is typically the communication bottleneck.

The propagation of a contribution to a sum is the core of PageRank's communication. To perform a propagation, the computation must read the graph adjacencies to identify which vertices are connected and subsequently access the contribution for the source vertex and the sum for the destination vertex. We categorize the memory traffic for this PageRank communication into "edge" traffic, which accesses the graph adjacency information, and "vertex" traffic, which accesses the contributions or sums associated with vertices. The edge traffic typically enjoys good spatial locality, as most implementations process the neighbors of a vertex consecutively and many graph layouts (e.g., CSR) store neighbor identifiers continuously. The vertex traffic has the potential to have much lower locality, since a vertex could potentially be connected to any other vertex, and so the vertex value accesses are unlikely to be consecutive. For graphs with many vertices, the corresponding arrays that associate a value with each vertex are much larger than the cache, so non-consecutive accesses to these arrays are likely to have poor cache locality.

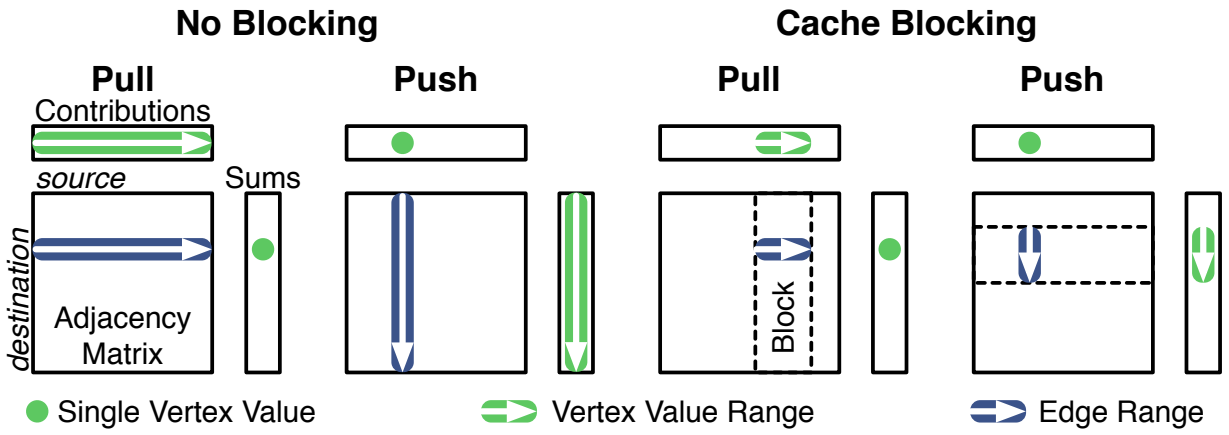

Figure 7.3: Accesses needed to process one vertex and its neighbors for PageRank, with or without 1D cache blocking for both pull (row-major) and push (column-major) directions. All approaches enjoy high locality when accessing the adjacency matrix (edge traffic). The vertex value arrays (contributions or sums) are much larger than the cache size, and thus accessing a sparse range within them could have low locality. Without blocking (left), one vertex value access obtains high locality at the expense of the other. Pull has high temporal locality on the sums, but low locality reading the contributions. Push has high locality reading the contributions, but low locality for the sums. Cache blocking (right) reduces the range of vertex values to the current block in order to improve locality.

Vertex traffic is composed of accesses for the contributions as well as accesses for the sums, and it is challenging to improve locality for both access streams simultaneously. One of the vertex value accesses will have high temporal locality (probably even register allocated), but the other will have potentially low locality as there is no restriction on a vertex's location relative to its neighbor (Figure 7.3). For example, in the pull direction, the sum of incoming contributions will have high locality, but reading the neighbor's contributions could have low locality. For the push direction, the outgoing contribution will have high locality, but reading (and writing) its neighbors' sums could have low locality. The low locality vertex value accesses not only increase the number of memory requests (low temporal locality), but the low locality can also result in unused words within transferred cache lines (poor spatial locality). These unused words are problematic, as they waste bandwidth and energy.

Since the vertex values are stored contiguously in arrays, the graph labelling (or "layout") has a tremendous impact on the locality of the vertex value accesses. For accesses that will potentially have low locality, whether they actually experience low locality is determined by the graph's layout. An ideal high-locality graph layout has all of its non-zeros in a narrow diagonal when viewed by its adjacency matrix. Not only does this improve the spatial locality for processing one vertex, since all of its neighbors' vertex values will be adjacent, but it also improves the temporal locality between vertices because there will be large overlaps between their neighbors. Unfortunately, some graphs' topologies make it difficult to find such an ideal

layout. These graphs of interest are often low-diameter and are often social networks [145]. Alternatively, there may be situations in which the time to compute and transform the graph into such a layout is not warranted.

Cache blocking [123] is a technique to reduce the negative impact of low-locality vertex value accesses. By partitioning the graph into blocks, the potential range for vertex values is reduced sufficiently such that the corresponding array segments for the vertex values are small enough to reside in cache, thus improving the locality of that access stream (Figure 7.3). Unfortunately, this gain for the low-locality access stream comes at the expense of the high-locality access stream. For example, blocking in the pull direction will improve the locality of reading the neighbors' contributions, but worsen the locality of accessing the sum, as the sum must be re-read and written for each block. The more blocks the graph is partitioned into, the more times the sums must be re-accessed. The block size is a tradeoff between the localities of the two access streams, with the optimum size resulting in moderate locality for both streams. Cache blocking can be done in one or two dimensions and can be applied to either direction (push or pull). It is also worth distinguishing cache blocking for sparse matrices from register blocking for sparse matrices, which is often done to reduce the amount the index array is read [123].

To implement cache blocking, the graph data structure needs to be modified to accomodate easy access to each block. One technique is to store the graph blocks each as their own graph (in CSR). If the graph is sufficiently sparse (e.g. expected number of nonzeros per row per block < 1), it may be advantageous to store each block as an edge list instead to cut down on the index traffic.

The challenge for PageRank communication is that it is difficult for both vertex value memory access streams to obtain high locality simultaneously. With the push technique or the pull technique, one stream does well at the expense of the other. Cache blocking allows for a more continuous locality tradeoff between the two streams, but it does not allow for them both to have high locality simultaneously.

7.4 Propagation Blocking

To improve the low locality vertex value accesses, we propose *propagation blocking*. Unlike cache blocking that blocks the graph itself, we block the propagations. Propagation blocking stores the propagation to memory in a semi-sorted manner, so when the propagations are read later, they will have better locality. We split the propagation of contributions for the push technique into two phases: *binning* and *accumulate* (Figure 7.5). In the binning phase, all of the contributions are read, but instead of adding them directly to their sums, we insert them into bins corresponding to their destination vertices. Each bin corresponds to a contiguous range of vertices, but the number of bins is small enough (e.g., 64) that the insertion points of these bins can fit in cache simultaneously. This binning phase is analogous to a radix partitioning done for a database join or a sorting kernel. When inserting a contribution into

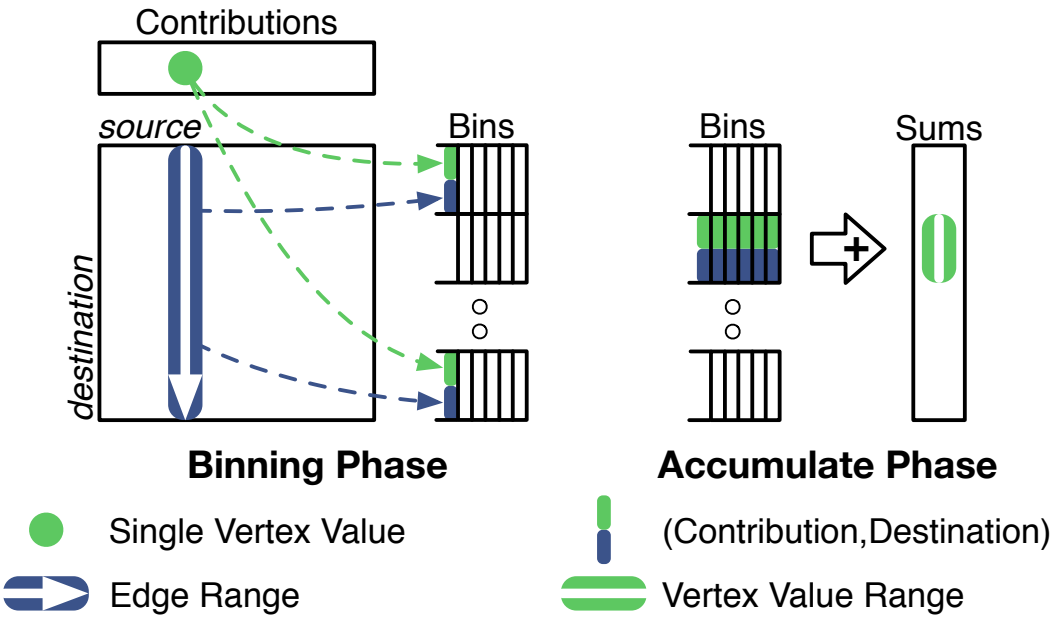

Figure 7.4: Propagation blocking. In the binning phase, vertices pair their contributions with their destination indices and insert them into the appropriate bins. In the accumulate phase, the contributions are reduced into the correct sums.

one of these bins, the destination vertex identifier is also included to create (*contribution*, *destination*) pair.

In the accumulate phase, one bin is processed at a time and the contributions are added to their appropriate sums (Figure 7.4). Because the bin corresponds to a reduced range of vertices, all of the corresponding sums should fit in cache at once. Reading the (contribution, destination) pairs from the bin will also enjoy high spatial locality since they will be contiguous.

Propagation blocking reduces communication through the use of additional memory space to store the propagations. The amount of additional memory required can be substantial, since each directed edge in the graph will need space for two words (contribution and destination). The additional memory is split into bins, and the number of bins is the number of vertices in the graph divided by the bin width. The bin "width" is chosen such that the number of vertices associated with each bin is small enough that their corresponding slice of the sums array can fit in cache.

To further reduce communication, propagation blocking can leverage a deterministic layout for the bins. If the locations that contributions are written to within a bin are always the same for multiple invocations on the same graph, the destination identifiers can be stored in a separate data structure. This halves the number of writes during the binning phase, as only the contributions and not their destinations need to be written. During the accumulate phase, the bin is read in lockstep with the destination indices to determine the destinations

```
scores[:] ← 1 / |V|
sums[:] ← 0
bins[:] ← {}
base ← (1 - d) / |V|
for i until max_iters do
  for u ∈ V do
    contribution ← scores[u] / |N+(u)|
    for v ∈ N+(u) do
      bins[v/num_bins] ← bins[v/num_bins] + (contribution, v)
    end for
  end for
  for b ∈ 1 ... num_bins do
    for (contribution, v) ∈ bins[b] do
      sums[v] ← sums[v] + contribution
    end for
    bins[b] ← {}
  end for
  for u ∈ V do
    scores[u] ← base + d × sums[u]
    sums[u] ← 0
  end for
end for
```

Figure 7.5: PageRank by propagation blocking

for each contribution. The separate arrays containing the destination indices for the bins can be reused after the first iteration of PageRank or even computed in advance.

Propagation blocking performs more load and store instructions than an unblocked PageRank in order to insert propagations into the bins and read propagations back from the bins. Since propagation blocking makes use of every word in each transferred cache line, it can transfer fewer cache lines to achieve a net memory communication reduction. Whether propagation blocking is advantageous depends on how well the baseline makes use of every word in each cache line it transfers.

7.5 Communication Model

We present simple analytic models for the amount of communication each PageRank implementation strategy performs in order to gain qualitative insights into their tradeoffs. Our models assume a uniform random input graph due to its simplicity, but we acknowledge a random graph is the worst case for many of these implementations due to its lack of locality. For communication volume, we use units of cache lines since that is the unit of transfer with

main memory. Within cache lines we use the unit of words (e.g., 32-bit words). We model only a single global iteration of PageRank, since each implementation strategy we model communicates the same amount each iteration. We use the following parameters:

n number of vertices ($|V|$)

k average directed degree ($nk = |E|$)

b number of words per cache line

c number of words in cache

r number of graph blocks for cache blocking

Pull Baseline

To read the graph from a CSR layout requires kn/b cache lines to read the adjacencies and $2n/b$ cache lines to read the indices. Our index uses 64-bit pointers to support greater than 4 billion edges, so we count each of these pointers as two words. To read the vertex values, looking up the destination sums requires reading n/b cache lines. The source contributions could potentially be in cache. We approximate the cache hit rate as $1-c/n$ (assuming $n > c$) and since there are kn directed edges, reading the contributions requires $(1 - c/n)kn$ cache lines. To output the final scores requires writing n/b cache lines. Altogether, the number of cachelines the pull technique reads is:

$$\left(\left(1 - \frac{n}{c}\right) + \frac{1}{b} + \frac{3}{kb}\right) nk$$

Cache Blocking

We model the communication for 1D cache blocking in the push direction using a CSR data structure for each block. Reading the adjacencies still requires reading kn/b cache lines, however, with r blocks we will also read r index arrays, so reading in all of the graph requires a total of $(k + 2r)n/b$ cache line reads. Presumably, the blocks are small enough such that $n/r < c$, so the only vertex value traffic needed is compulsory, but the blocked vertices will be re-read, yielding $(r + 1)n/b$ cache line reads. To output the final scores requires writing n/b cache lines. Altogether, the number of cachelines 1D cache blocking on CSR requires reading is:

$$\frac{(k + 3r + 1)n}{b}$$

If the graph is sufficiently sparse ($k < 2r$), using an edge list to hold each block instead of CSR reduces the index communication, yielding a cacheline read total of:

$$\frac{(2k + r + 1)n}{b}$$

We do not model 2D cache blocking since in our context, 2D cache blocking will not communicate significantly less than 1D cache blocking. As 2D cache blocks are processed temporally, they will effectively merge into a 1D cache block along the dimension they are being processed along. This effective 1D cache block will communicate the same as if it were truly a 1D cache block.

Propagation Blocking

We assume there are an adequate number of blocks such that $n/r < c$. Like the pull case, reading the CSR graph requires reading $(k+2)n/b$ cache lines. Reading the source vertex values requires n/b cache line reads and outputting the final scores requires n/b cache line writes. Propagating contributions writes $2kn/b$ cache lines in the binning phase and reads $2kn/b$ cache line in the accumulate phase. Reusing the destination indices saves kn/b writes from the binning phase. Altogether, the number of cachelines propagation blocking reads is:

$$\left(3 + \frac{3}{k}\right)\frac{kn}{b}$$

and writes (when reusing destination indices):

$$\left(1 + \frac{1}{k}\right)\frac{kn}{b}$$

Commentary

For the pull technique, the miss rate $(1 - c/n)$ strongly impacts the amount of traffic. Comparing the total communication of propagation blocking to the pull technique, we see propagation blocking will be advantageous when:

$$b \geq \frac{3}{1 - c/n}$$

This tradeoff is intuitive, as the opportunity for propagation blocking is if the pull technique frequently misses the cache.

Propagation blocking is advantageous to cache blocking using an edge list when:

$$r \geq k + \frac{3}{2}$$

To first order, the amount of traffic for cache blocking is proportional to r, while for propagation blocking it is proportional to k. These dominant factors correspond to the attribute each technique is blocking. Cache blocking breaks up the graph, and r is proportional to n/c. For larger graphs, cache blocking will use more blocks and reload the vertex values more times, which will decrease its communication efficiency. Propagation blocking breaks

up the propagations, which are proportional to k. Thus, propagation blocking will not have a change in communication efficiency for larger graphs. From our simple models, we see propagation blocking will communicate less than cache blocking when the graph is sparse enough and has sufficiently more vertices than can fit in the cache.

7.6 Evaluation

To perform our evaluation, we use the suite of sparse, low-diameter graphs in Table 7.1 and we provide more background on them in Section 2.3. Each graph uses the vertex labelling provided by its original data source, which is often chosen intelligently. The graph webbase has an optimized vertex labelling, and to show the benefit of that labelling, we randomize webbase's labelling to produce webrnd. Since PageRank computation is proportional to the number of directed edges, in this chapter we use the directed degree since we find it to be a more instructive metric. The directed degree of an undirected graph is twice its average degree.

For consistency, we start all of our implementations from the same codebase, and we use our GAP Benchmark Suite's reference implementation of PageRank (Section 4.4). We use the following implementations:

Baseline is the reference implementation and it computes PageRank in the pull direction.

Cache Blocking (CB) improves our baseline by performing 1D cache blocking. It computes PageRank in the push direction, and it uses a CSR data structure for each block.

Propagation Blocking (PB) implements our propagation-blocking technique in the push direction.

Deterministic Propagation Blocking (DPB) improves upon PB by using the optimization of storing the destination indices separately so that during the binning phase, only the contributions need to be written.

For our blocking implementations, we first tune the block width and then compute the number of blocks based on the width. After testing many block widths, we determined our implementations performed best when the corresponding vertex value array segments are 2 MB. We explore the bin width tradeoffs for propagation blocking later in this section. Each result in this section is the average of multiple trials of a single iteration of PageRank. We do not include the time to block the graph for CB or to allocate the bins for PB, as these can be done in advance or reused for other algorithms.

To perform our evaluation, we use a dual-socket Intel Ivy Bridge server (IVB), similar to what one would find in a datacenter (Table 7.2). To access hardware performance counters, we use Intel PCM [81]. We compile all code with gcc-4.8, except the external baselines that use Cilk from icc 14. To ensure consistency across runs, we disable Turbo Boost (dynamic

Graph	Description	Vertices (M)	Edges (M)	Directed Degree	Directed
Urand	uniform random [58]	134.2	1,073.7	16.0	
Kron	Kronecker generator [66, 94]	134.2	1,062.8	15.8	
Citations	academic citations [149]	49.8	949.6	19.0	✓
Coauthors	academic coauthorships [149]	119.9	646.9	10.8	
Friendster	social network [170]	124.8	1,806.1	28.9	
Twitter	social network [91]	61.5	1,468.3	23.8	✓
WebBase	2001 web crawl [47]	118.1	632.1	5.4	✓
WebRnd	WebBase randomly relabelled [47]	118.1	632.1	5.4	✓

Table 7.1: Graphs used for evaluation. All of the graphs come from real-world data except kron and urand. The kron, urand, and twitter graphs are also in our GAP Benchmark Suite.

Architecture	Ivy Bridge EP
Intel Model	E5-2667 v2
Released	Q3 2013
Clock rate	3.3 GHz
# Sockets	2
Cores/socket	8
Threads/core	2
LLC/socket	25 MB
DRAM Capacity	128 GB
DRAM Type	DDR3-1600

Table 7.2: IVB system specifications

voltage and frequency scaling). Our experiments are single-threaded to get precise communication measurements. All of the implementations could be parallelized, but the focus of this chapter is on communication.

Baseline Validation

To ground our work, we validate the performance of our baseline implementation by comparing it to three established codebases. We use the PageRank implementations from Galois (pull version) [122] and Ligra (non-delta version) [146]. We also use the Compressed Sparse Blocks (CSB) SpMV implementation, but since it does not perform the additional computations necessary for PageRank, our measurements overestimate its performance [30].

Codebase	Time (s)	Memory Reads (M)	Reads / second (M)	Instructions Executed (B)
Baseline	42.6	4,442	104.3	33.6
CSB (SpMV)	81.9	5,448	66.5	116.9
Galois	100.6	4,946	49.2	82.0
Ligra	129.5	13,180	101.8	114.4

Table 7.3: PageRank iteration on 128M vertex 2B undirected edge uniform random graph

Table 7.3 presents the performances of our baseline and the three established codebases using a single-thread to process a uniform random graph of 128M vertices and 2B undirected edges (directed degree of 32). Although our baseline implementation is simple, its litheness allows it to communicate the least and execute the fewest instructions while still using the most memory bandwidth. Both our baseline and Ligra obtain nearly full memory bandwidth utilization (we achieve a maximum of 108 M memory requests/second with synthetic microbenchmarks), but Ligra performs 3× the total communication. CSB and Galois both execute so many additional instructions that their memory bandwidth utilization is bottlenecked by the instruction window size, as we discussed in Chapter 5. Overall, our baseline implementation is substantially faster than prior work, so any performance improvements over our baseline represent very substantial improvements over prior work.

Quantifying the Opportunity for Propagation Blocking

To gauge the opportunity for how much our blocking techniques can improve locality, we first measure how much locality there is in our benchmark graphs. In theory, the amount of memory communication for the vertex traffic and the edge traffic (reading the graph) should be equal, but if the vertex value accesses have low locality, the vertex traffic can consume much more than half of the memory traffic. Figure 7.6 shows that most of our input graphs have low-locality layouts, as the edge traffic consumes far less than the expected 50%. To measure the fraction of edge memory traffic, we process each graph twice: once with our baseline implementation, and once only reading the graph. The traffic we measure for reading only the graph is also in close agreement with our model from Section 7.5.

The detailed results of our baseline in Table 7.4 shows the impact of a graph's layout and topology on communication volume. The webbase and webrnd graphs have exactly the same topology, but webbase's optimized layout enjoys many more cache hits for its vertex accesses, which in turn reduces the total number of memory requests. This impact is also visible in Figure 7.6, as the same number of accesses to read the webbase graph now constitute a much larger fraction of the memory traffic. Although the kron graph is the same size as the urand graph, its power-law degree distribution improves the temporal locality of vertex value accesses, and so it too enjoys more cache hits, reducing its vertex traffic. Overall, all of our input graphs except webbase have low locality and could thus be amenable to blocking.

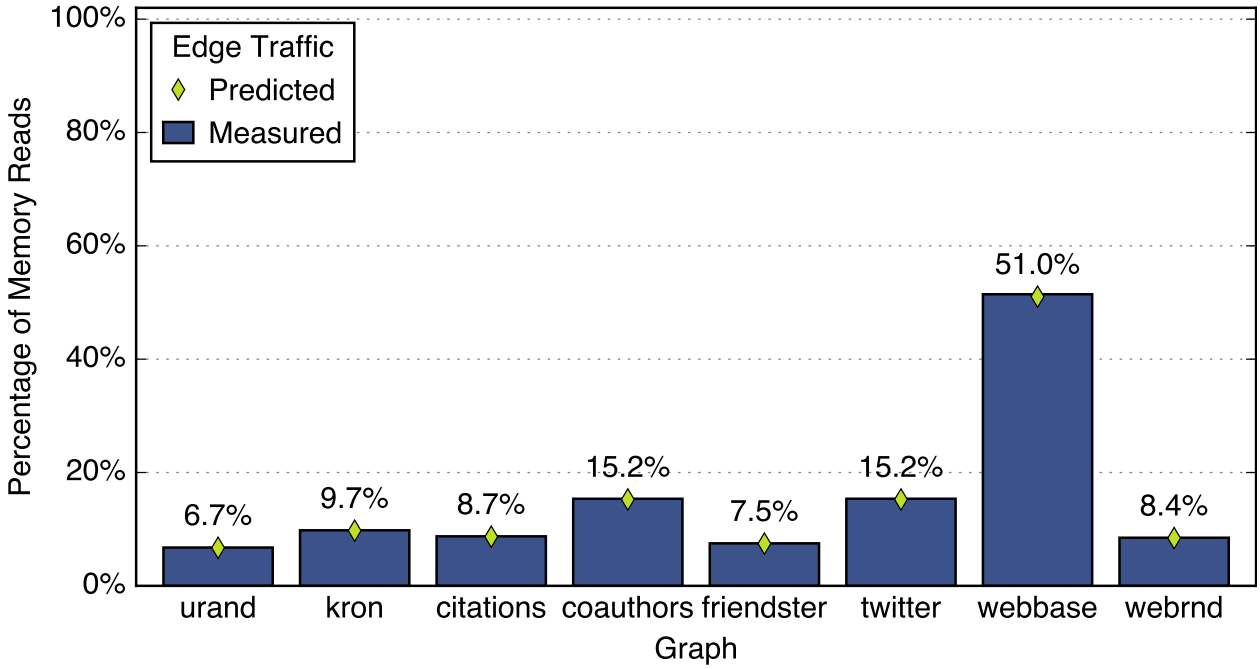

Figure 7.6: Fraction of read memory traffic to read graph from CSR

	Baseline				DPB			
	Time	Memory (M)		Instructions	Time	Memory (M)		Instructions
Graph	(s)	Reads	Writes	Exec. (B)	(s)	Reads	Writes	Exec. (B)
urand	21.8	2,247.6	33.6	19.9	13.6	458.4	164.1	77.6
kron	17.7	1,536.8	32.1	18.7	13.7	454.5	163.9	79.6
citations	7.9	756.5	12.7	8.0	5.6	199.0	70.6	33.8
coauthors	8.5	628.8	26.4	12.2	8.0	291.3	106.6	48.3
friendster	33.4	3,237.9	36.3	28.1	21.4	732.7	256.2	129.0
twitter	8.6	652.5	15.2	11.0	8.0	301.4	105.8	51.7
webbase	2.2	106.4	23.4	7.7	3.8	164.7	63.1	25.9
webrnd	7.2	644.5	25.3	8.5	5.2	165.0	63.5	26.2

Table 7.4: Detailed performance results for baseline and DPB

Comparing Blocking Approaches

All three blocking implementations deliver substantial performance (Figure 7.7) and communication (Figure 7.8) improvements relative to our baseline. The reductions in communication are greater than the reductions in execution time, because our baseline implementation utilizes more memory bandwidth. DPB's optimization of reusing destination indices does provide a modest improvement over PB. In terms of execution time, DPB and CB are competitive with each other.

The detailed results in Table 7.4 show that DPB greatly reduces the amount of read traffic, but its overall communication reduction is reduced by the additional write traffic to store propagations to bins. To perform propagation blocking, DPB performs on average four times as many instructions as the baseline, and this reduces DPB's memory bandwidth utilization.

We use the GAIL metrics (Chapter 6) to visualize the locality-bandwidth tradeoff the blocking implementations experience (Figure 7.9). Since all implementations process the same number of edges, we do not display the number of traversed edges. For the GAIL metrics shown, moving towards the origin represents faster edge processing, since moving to the left represents better cache locality, and moving down represents greater memory bandwidth utilization. The baseline implementation generally obtains high memory bandwidth utilization, but it has a wide range of memory requests per edge caused by the varying localities of the input graphs. All three blocking implementations (CB, PB, and DPB) generally improve performance because they improve locality (farther to left) by more than they worsen memory bandwidth (farther up). The propagation blocking implementations (PB and DPB) have less variation in their performance since their communication is less sensitive to topology. The reduced performance variation makes the performance of the propagation blocking implementations more predictable.

Comparing DPB to CB on each graph (Figure 7.7 and Figure 7.8), DPB does succeed in reducing communication more often than CB. Since both CB and DPB are slower than the baseline on webbase, we focus on the remaining graphs that have worse layouts. Of those remaining seven graphs, DPB moves less data on five of them. Those two graphs where CB communicates less have the fewest vertices and are amongst the densest. Since our CB implementation uses a constant block width, a graph with fewer vertices will use fewer blocks, which will result in fewer times the vertex values are reloaded (less communication). Additionally, the increased density gives CB more useful work to amortize the reloading of the vertex values.

To quantify the topological properties necessary for blocking to be advantageous, we artificially control locality by generating graphs of the same degree but with varying numbers of vertices (Figure 7.10). With fewer vertices, the vertex values are more likely to remain in cache. We normalize these communication volumes to the number of edges in the graph (second GAIL metric). For our baseline, once the graph becomes sufficiently large, it overflows the cache and moves more data. For CB, the number of blocks increases along with the number of vertices, which causes the vertex values to be re-read more times. The constant

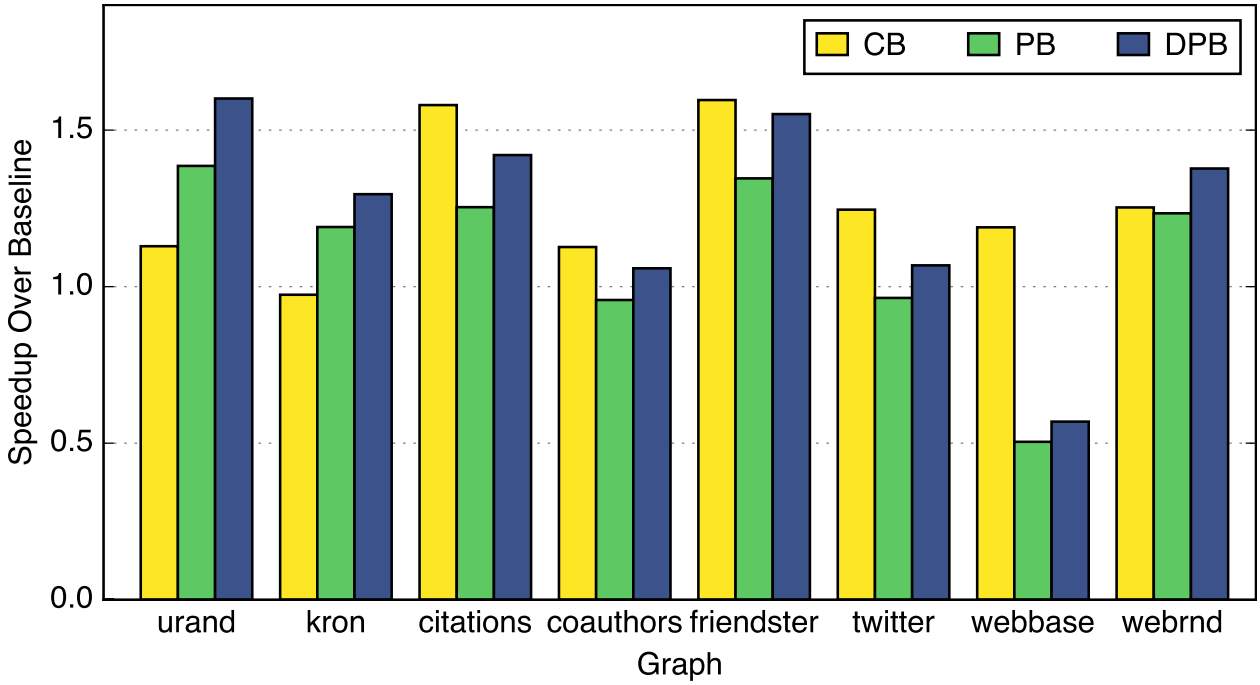

Figure 7.7: Execution time improvement over baseline

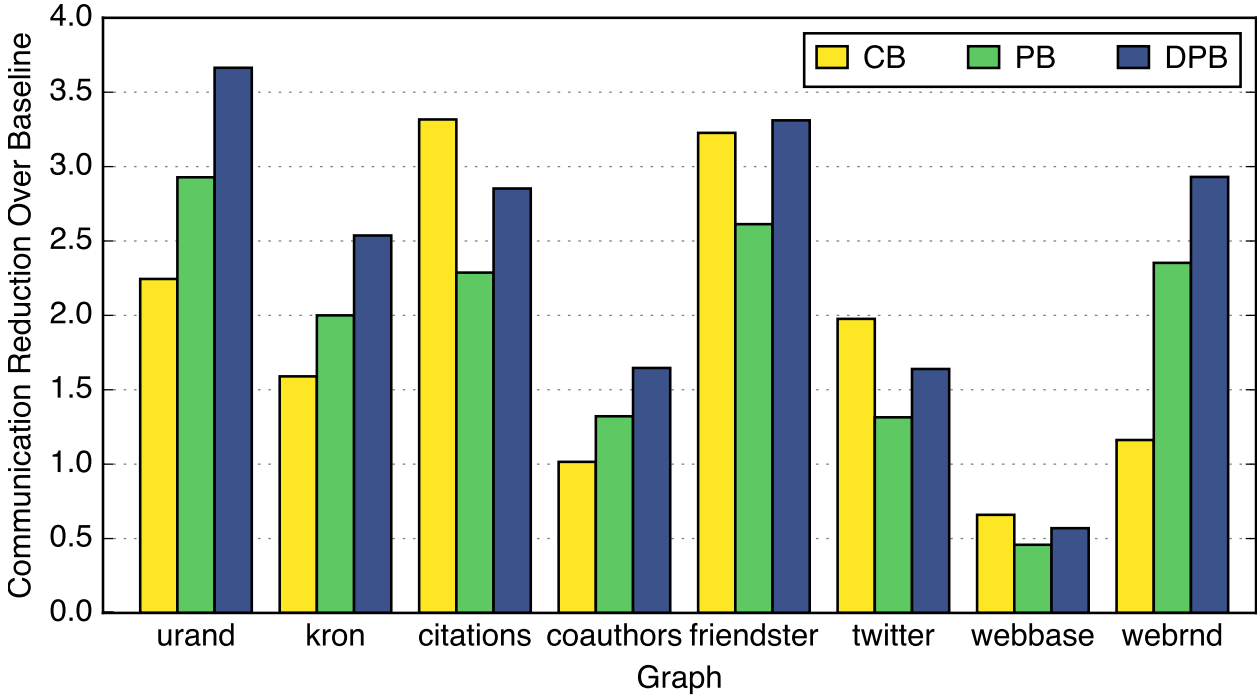

Figure 7.8: Communication reduction over baseline

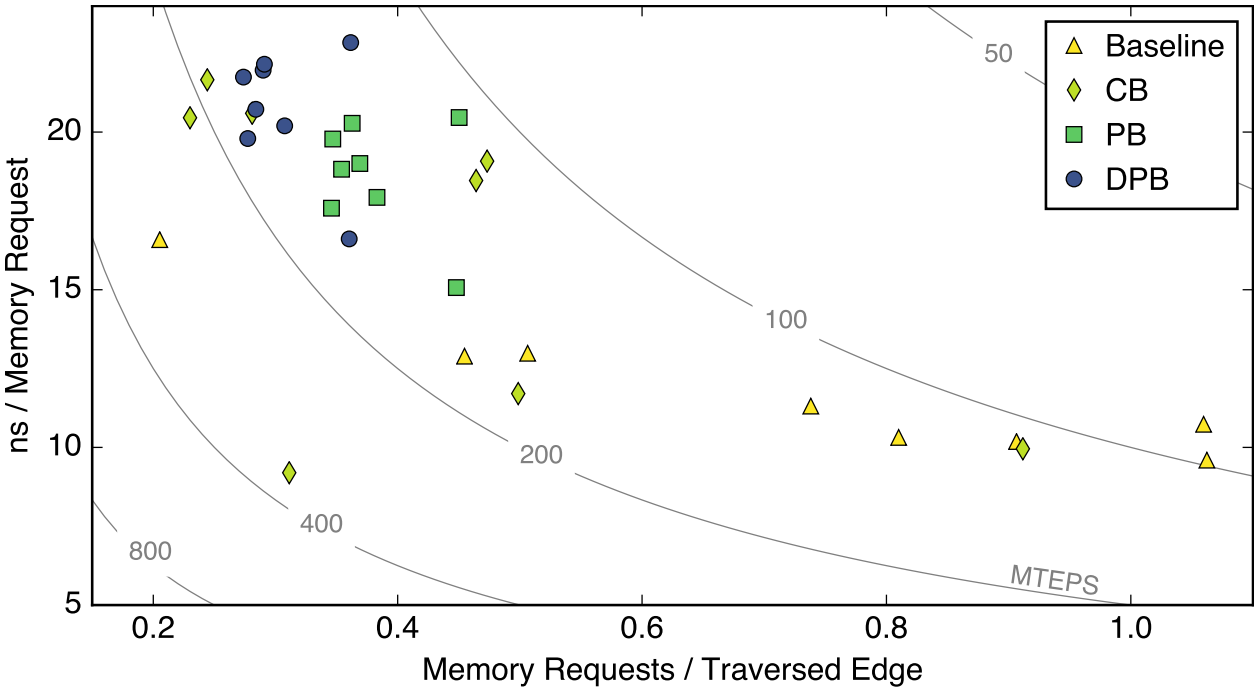

Figure 7.9: GAIL metrics across benchmark suite. Traversed edge totals not shown since equal for all implementations. Contours represent millions of traversed edges per second (MTEPS), so processing rates increase toward the origin.

number of memory requests per edge for DPB indicates that the number of edges is the primary determinant of memory traffic, and it is not the expected hit rate (c/n) like the baseline, or the number of blocks r (proportional to n/c) for CB. Thus, which approach communicates the least depends on the number of vertices relative to the cache size. For the smallest graphs, blocking is unmerited and our baseline is the most communication efficient. For mid-size graphs, cache blocking is the most efficient, but as the graph gets too large, the overhead of reloading the vertex values makes it less efficient. For the largest graphs, propagation blocking provides the most scalable communication.

In Figure 7.11 we vary the degree to find the sparsity for which DPB is advantageous to CB. Since all of the graphs have the same number of vertices, CB will use the same number of blocks. For denser graphs, CB will have more useful work to amortize the compulsory traffic of reloading vertex values for each block, and thus it becomes more efficient on average. In this experiment with a 128 million vertex uniform random graph, DPB will communicate less than CB if the directed degree is 36 or less. For graphs with more vertices, that degree cutoff will be higher, since CB will have more compulsory traffic to amortize.

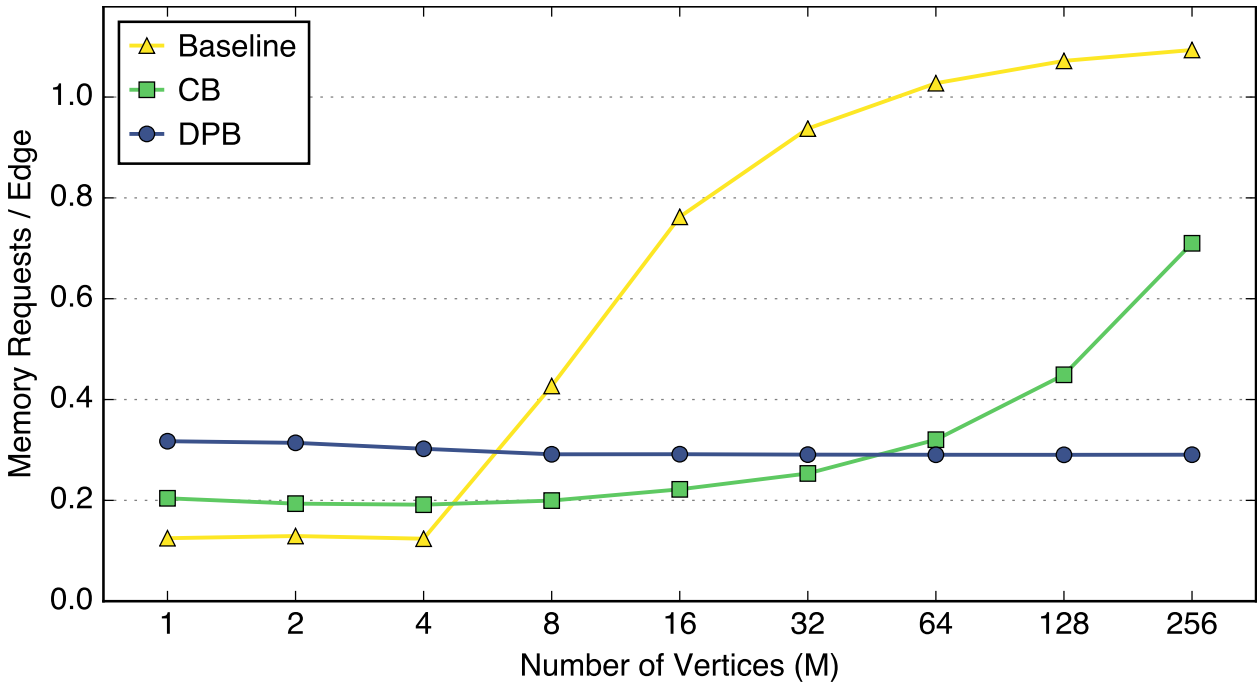

Figure 7.10: Communication efficiency for varying number of vertices of a degree=16 uniform random graph

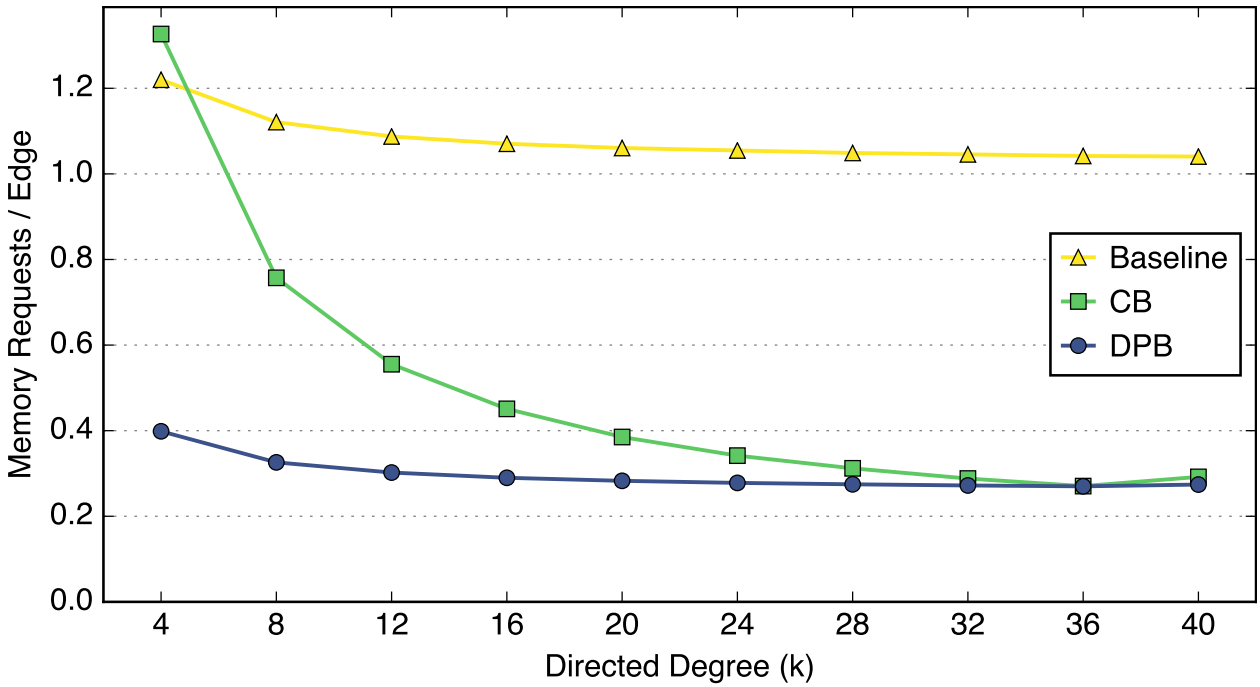

Figure 7.11: Communication efficiency for varying degree of a 128M vertex uniform random graph

Selecting Bin Size

We vary the bin width to determine its impact on propagation blocking's performance. Once the vertex value array partitions that correspond to each bin are small enough to fit in the cache, there is not much change in communication volume (Figure 7.12). The webbase graph is insensitive to bin width since its high-locality layout obviates blocking. Once memory communication is minimized, there is additional execution time benefit to using slightly smaller bins (Figure 7.13). However, making the bins too small makes them too numerous, which causes more L1 cache misses for bin insertions during the binning phase. For our platform, we select a partition size of 2 MB, as it is typically the fastest while communicating little.

7.7 Implementation Considerations

To improve the performance of our propagation-blocking technique, we use a number of optimizations, some of which are platform specific (x86/Linux). Our propagation-blocking implementation is written in C++, and it performs low-level memory management and makes use of compiler intrinsics for special instructions. Low-level programming is not a requirement to use propagation blocking, and propagation blocking can be encapsulated in a framework to hide its implementation details from a graph algorithm implementor. In Appendix B, we describe a high-level language for implementing graph algorithms that uses propagation blocking internally.

To improve the performance of the binning phase, we restrict our bin widths to powers of two so we can use a shift instructions to quickly compute a destination bin (instead of an integer divide).

To reduce the amount of communication during the binning phase, we use non-temporal stores and other optimizations from prior work on radix partitioning [133, 140, 159]. When writing to the bins, even though we are only writing, the processor will often read the cache line from memory first before overwriting it in cache (write allocate). By using Intel's non-temporal (streaming) store instructions, we instruct the processor to bypass the cache when performing the writes, which obviates the read from memory for the write allocate [80].

Non-temporal store instructions on their own can be wasteful, as they write only part of a cache line but still transfer a full cache line. To reduce this waste, we coalesce our writes together in software by using small cache line-aligned fixed-size buffers. We store into the buffers with normal store instructions, and since they are small, they reside in cache. When a buffer becomes full, we copy it in bulk to its corresponding bin in memory using the non-temporal store instructions. To implement our copy routine, we obtain the best performance using the AVX non-temporal stores, but we also experimented with the SSE2 streaming stores and the ERMSB idiom [80].

We use software prefetch instructions to improve memory bandwidth utilization during the accumulate phase. In particular, we prefetch ahead within the bin being currently

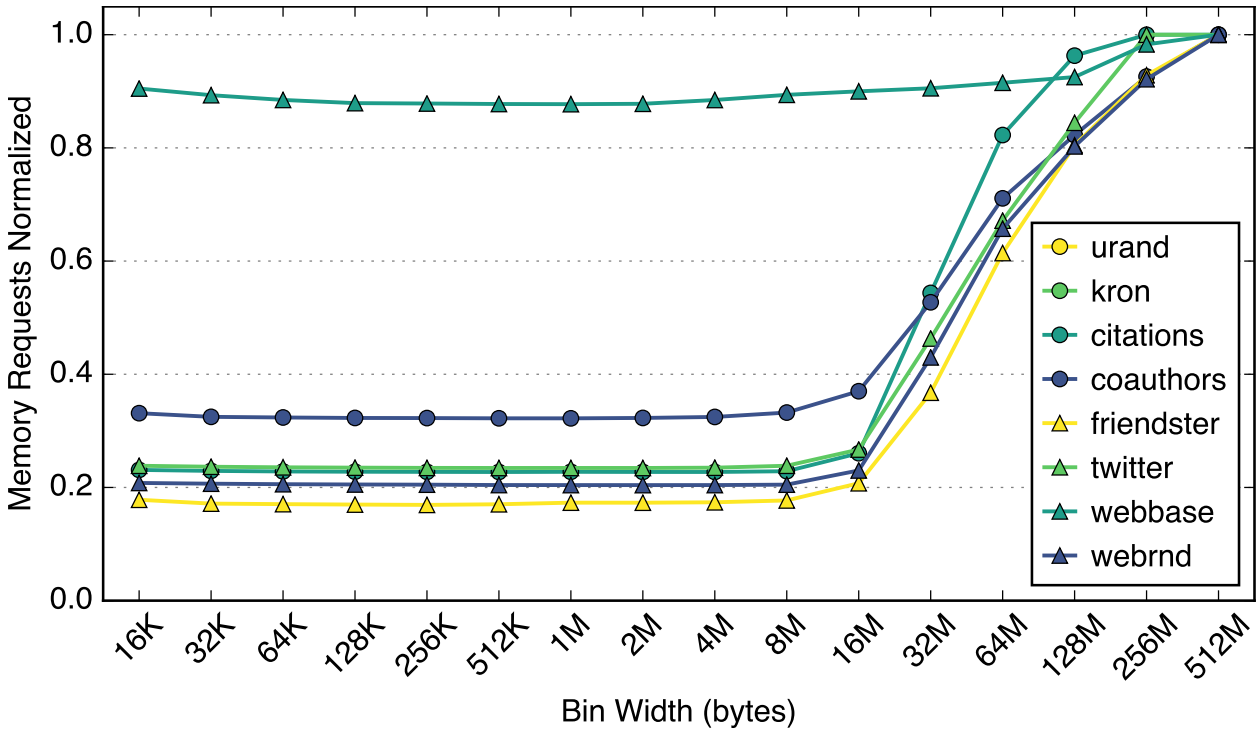

Figure 7.12: Bin width impact on total memory communication for DPB by graph

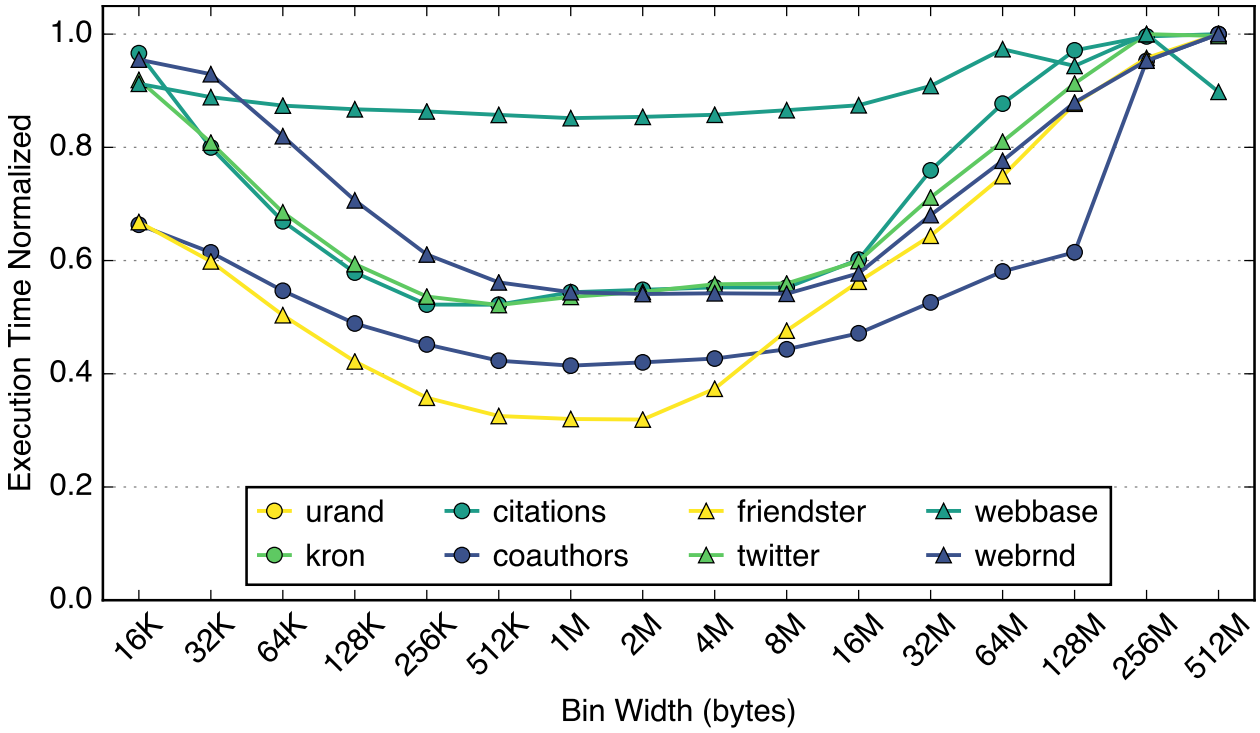

Figure 7.13: Bin width impact on execution time for DPB by graph

processed. Although that access stream is continuous and should be detected by the hardware prefetcher, we speculate that the other loads and stores for accessing the sums confuses the prefetcher. We attempted to use software prefetch instructions in other places, but were unsuccessful in obtaining an improvement.

Although we do not care about the contents of the bins prior to using them, we still find it is important to initialize them. If left the bins are left uninitialized, we observe Linux zeroing out the contents of the corresponding memory pages during our program's first access, which substantially increases the amount of write traffic. Due to lazy memory allocation, although our program has allocated the memory space, Linux has not yet given our program access to physical memory pages. To fix this problem, as soon as we allocate the bins, we stride through the bins to trigger Linux to zero the pages. Since Linux is performing the zeroing, we only need to stride at an interval equal to the page size.

7.8 Related Work

SpMV is communication-bound, and as a consequence, much of the prior work on optimizing for performance has naturally also optimized communication [162]. Bender et al. provide complexity bounds on the amount of communication for SpMV [21], and their I/O model matches our model of a cache and main memory. Furthermore, our empirical results confirm their side result that a uniform random sparse matrix requires approximately half the I/Os of their modeled worst case. Nishtala et al. provide criteria for when cache blocking will be advantageous for SpMV, and it is consistent with our results [123]. For example, we find the number of vertices (x in their formulation) and the randomness of the matrix to be important requirements.

There has been extensive prior work on reordering graphs and sparse matrices in order to improve locality. The Cuthill-McKee [46] technique and its follow-on RCM [62] apply BFS as a heuristic to find a good ordering. Shun compares multiple reordering techniques including DFS on social network topologies [145]. Compressing graphs exposes many of the same locality challenges, and the WebGraph framework is specifically designed to compress web crawls [28]. Unfortunately, no reordering technique is beneficial for all input graphs [145].

Relabelling the graph transforms the graph spatially, but the graph can also be transformed temporally. Changing the order in which edges are read from the graph can improve locality. Cache-oblivious algorithms use space-filling curves to obtain reasonable locality without any knowledge of the size of the cache, and they have been successfully applied to SpMV [176] and PageRank [105]. Unfortunately, these temporal transformations can greatly complicate parallelization.

7.9 Conclusion

As demonstrated by our benchmark graph webbase, when a high locality graph layout is available, communication for PageRank is naturally reduced. Unfortunately, such high locality graph layouts are not always available, which not only results in more cache line reads from memory, but also fewer words used per transferred cache line.

Blocking is a technique used throughout computing to reduce communication by improving locality. Blocking improves locality by reducing the working set of the application to be small enough to fit in the fast memory. The amount of communication for PageRank for conventional cache blocking is primarily determined by the number of vertices, as this determines the number of blocks and thus the number of times the vertex values must be re-read. By contrast, the amount of communication for our proposed propagation-blocking technique is primarily determined by the number of edges. From the linear algebra perspective, propagation-blocking communication is proportional to the number of non-zeros in the sparse matrix, while conventional cache blocking communication is proportional to the matrix dimensions. Thus, if the graph is sufficiently sparse, or if the number of vertices is sufficiently large, propagation blocking will be more communication efficient than cache blocking.

Our results demonstrate our propagation-blocking implementation can reduce communication substantially, but there are cases when our other implementations communicate less. The locality of the graph determines whether one should use the pull baseline (high locality) or either blocking approach (low locality). Unfortunately, a graph's locality is not easy to measure quickly, but hopefully from the context of the application there should be hints as to its locality. The degree and the number of vertices determine whether one should use propagation blocking (lower degree and more vertices) or cache blocking (higher degree and fewer vertices). Fortunately, those topological parameters are easy to access and the decision to use propagation blocking or cache blocking could be done dynamically at runtime.

In this chapter, we compare the time it takes to perform PageRank once the graph has been loaded and optimized, but the time to optimize the graph is also worth considering. It may be worthwhile to optimize the graph less if the reduction in graph preprocessing time is greater than the increase in kernel execution time. Relabelling the graph to improve its locality can be a time consuming optimization, as once the new mapping is selected, there will be extremely poor locality during the conversion. Partitioning the graph for cache blocking is less time consuming, as the conversion can be done in a streaming fashion in order to obtain good spatial locality. Creating the bins needed for propagation blocking is the least demanding optimization, as it only requires allocating memory. The bins could also be reused for other graphs that are not bigger than the original graph. Since propagation blocking requires comparatively less graph preprocessing, its use is beneficial for a greater number of situations since it requires less execution time to amortize.

In this work we target PageRank, but the propagation-blocking technique could easily be generalized to other applications. For graph algorithms, it is applicable whenever the algorithm considers vertex values on both sides of an edge. The benefit of propagation

blocking is dependent on the locality of the graph's layout. Propagation blocking could also be extended to handle generalized SpMV, and it should be beneficial as long as the matrix is sufficiently sparse.

Chapter 8

Conclusion

In this dissertation, we obtain insight into the fundamental factors that determine the speed of graph algorithms and use that understanding to improve performance. In this concluding chapter, we outline our main contributions and recommend future directions for the community to build upon our work.

8.1 Summary of Contributions

This dissertation makes the following main contributions:

1. Direction-optimizing breadth-first search, a novel algorithm that traverses low-diameter graphs faster by traversing only a small fraction of their edges (Chapter 3). This is the first algorithmic improvement to breadth-first search, as prior work only improves the efficiency of implementations and does not reduce the number of edges examined.

2. The Graph Algorithm Platform (GAP) Benchmark Suite to help the community improve graph processing evaluations via standard benchmarks (Chapter 4). We also provide a reference implementation that can be used as a high-performance baseline.

3. Workload characterization of graph processing on shared memory multiprocessors (Chapter 5). By using precise hardware performance counters, we demonstrate that contrary to prior conventional wisdom, most graph processing executions achieve moderate locality and are consequently typically not memory-bandwidth bound.

4. Graph Algorithm Iron Law (GAIL), a simple performance model to understand tradeoffs between a graph algorithm, its implementation, and its hardware execution platform (Chapter 6). GAIL incorporates the most important factors for graph algorithm performance: algorithmic efficiency, cache locality, and memory bandwidth utilization.

5. Propagation Blocking, an optimization technique to reduce memory communication of poor locality graph processing workloads (Chapter 7). Propagation blocking reduces communication by using additional memory capacity to improve spatial locality.

8.2 Future Work

We believe promising future work could be built on the two main thrusts of this work: understanding graph algorithm performance (analysis) and improving graph algorithm performance (synthesis). For many of the projects, we believe our GAP Benchmark Suite will be essential for evaluation and GAIL will help provide a deeper understanding of the results.

Analysis

Investigate Memory Hierarchy Performance for Graph Processing

One of the key contributions of our workload characterization is the empirical observation that graph processing workloads typically enjoy moderate cache locality and as a consequence, are unable to fully utilize the platform's memory bandwidth. Further investigation should analyze the properties of this cache locality. First, it would be interesting to measure the impact of cache size and organization on this locality for graph processing. In particular, it would be useful to determine whether parallel executions interfere constructively or destructively, and how this affects how much cache capacity should be allocated privately or shared. It would also be helpful to identify which accesses are most tolerant and most vulnerable to NUMA penalties. To make these measurements more instructive, it may be beneficial to categorize memory traffic in a manner similar to how we differentiate edge traffic from vertex traffic in Chapter 7. O'Neil et al. vary cache parameters in their graph workload characterization, but they focus on GPUs instead of shared-memory multiprocessors [125].

Characterize Dynamic Graph Processing Workloads

In this work we restrict ourselves to batch processing on immutable graphs, but there are also graph processing applications that operate on continuously changing real-world data. These dynamic graph processing workloads make changes to their graphs and attempt to compute on only the changes. By processing only the graph changes, their memory access patterns will be sparser and will probably experience worse spatial locality. However, because they are computing on recent changes, these changes could still be resident in fast memory due to temporal locality. It is not immediately clear which effect will be most impactful, so a proper evaluation is merited to be sure of the outcome.

Identify Bottlenecks for Distributed Graph Processing

Similar to how this work analyzes graph processing performance on shared memory multiprocessors, we believe an analogous analysis could be done for distributed graph processing. Distributed graph processing is often extremely inefficient (Figure 1.1), so identifying performance bottlenecks is a critical first step to improving performance. In particular, it would be useful to measure the network to observe not only the average bandwidth utilization, but whether network traffic is bursty, latency bound, or simply poorly balanced across compute nodes.

Synthesis

Online Determination of Δ-Parameter

The delta-stepping algorithm we implement for our benchmark reference implementation is the fastest SSSP algorithm, but the input-sensitivity of its Δ-parameter is problematic. Ideally, an online heuristic could determine Δ, and this heuristic should depend on the hardware platform and not the input graph, akin to the online heuristic we use to control our direction-optimizing BFS implementation. The wide range of reasonable values for Δ we observe in Figure 6.2 is encouraging, as the heuristic does not need to be perfect to be beneficial.

Hardware Support for Propagation Blocking

To further improve the performance of propagation blocking, hardware extensions could help eliminate bottlenecks. As we observe in Section 7.6, our speedups on IVB lag behind our reductions in communication because we utilize less memory bandwidth. Modifications to the processor to help it fully utilize its memory bandwidth could allow for the speedups to equal our communication reductions.

Profiling reveals that the binning phase is partially compute bound, as the memory bandwidth is only 60% utilized while IPC > 2. Hardware support for quick bin insertions or offloading copying could help by reducing the dynamic instruction count.

Traditional scatter and gather instructions available on the newest systems are promising (e.g., AVX-512 [80]), as they can trivially express the bulk of the accumulate phase. When looking up sums during the accumulate phase, we frequently miss in the L2 cache but hit in the L3 cache. These L2 misses are so frequent, that we are probably somewhat bottlenecked by the L3 bandwidth. High-performance scatter and gather instructions that directly access outer cache levels could use the on-chip bandwidth more efficiently.

Improve Out-of-Order Processors for Graph Processing

Since we find the size of the instruction window to be a memory bandwidth bottleneck, approaches to increase the window size or approximate a larger window (runahead) seem promising. A larger window size would clearly help out-of-order processors, but there is surely some size that will be limited by branch mispredictions and data dependencies. However, we don't know if that practical limit is 250 instructions or 500 instructions, so this merits further investigation.

For our workload, we observe IPCs typically below one even though the processor we evaluate is 4-wide. A narrower out-of-order core might have not much worse performance and could provide substantial area and energy savings. Despite having a low average IPC during our workload, there are almost certainly instances when the processor is able to burst and use its full width advantageously. In spite of this, to have a low average, the processor must spend much of the time executing instructions slowly or waiting, and a narrower core could do that just as well. The performance

loss from making the core narrower will almost certainly be less than the energy and area savings, resulting in a more efficient core. We speculate a 2-wide processor might be the most efficient design point [7]. Furthermore, the area savings from making the processor narrower could be used to increase the size of the instruction window.

Graph Processing Hardware Accelerators

In this work, we observe the difficulty scalar processors experience fully utilizing memory bandwidth for graph processing, so specializing hardware for graph processing may be the most tractable means to increase memory bandwidth utilization. There is prior work on graph processing hardware accelerators [4, 48, 124, 126], but we believe there is still room for improvement. In particular, we observe poor programmability to be a key limitation of the accelerators proposed so far. This programmability is not only the ability to execute different graph kernels on the same hardware instantiation, but to also provide the flexibility needed to implement leading optimized algorithms such as direction-optimizing BFS and delta-stepping. A slowdown in Moore's Law may make such graph processing accelerators inevitably the only means to further improve efficiency.

Bibliography

[1] Christopher R Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A relational engine for graph processing. *arXiv*, 1503.02368 [cs.DB], 2015.

[2] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. Scalable graph exploration on multicore processors. *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[3] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. CRONO: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. *International Symposium on Workload Characterization (IISWC)*, 2015.

[4] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. *International Symposium on Computer Architecture (ISCA)*, pages 105–117, 2015.

[5] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. *International Conference on Supercomputing (ICS)*, 1990.

[6] Wendell Anderson, Preston Briggs, C Stephen Hellberg, Daryl W Hess, Alexei Khokhlov, Marco Lanzagorta, and Robert Rosenberg. Early experience with scientific programs on the Cray MTA–2. *High Performance Computing, Networking, Storage and Analysis (SC)*, 2003.

[7] Omid Azizi, Aqeel Mahesri, Benjamin C Lee, Sanjay J Patel, and Mark Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis. *International Symposium on Computer Architecture (ISCA)*, 38(3):26–36, 2010.

[8] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. Four degrees of separation. *ACM Web Science Conference*, pages 45–54, 2012.

[9] David Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. *International Conference on Parallel Processing (ICPP)*, 2006.

[10] David Bader and Kamesh Madduri. SNAP: Small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks. *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2008.

[11] David A Bader, Guojing Cong, and John Feo. On the architectural requirements for efficient execution of graph algorithms. *International Conference on Parallel Processing*, Jul 2005.

[12] David A. Bader, John Feo, John Gilbert, Jeremy Kepner, David Koester, Eugene Loh, Kamesh Madduri, Bill Mann, and Theresa Meuse. HPCS scalable synthetic compact applications #2 graph analysis. version 2.2. `www.graphanalysis.org/benchmark`, 2004.

[13] David A Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. Approximating betweenness centrality. *International Workshop on Algorithms and Models for the Web-Graph*, pages 124–137, 2007.

[14] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286:509–512, Oct 1999.

[15] Scott Beamer, Krste Asanović, and David Patterson. Searching for a parent instead of fighting over children: A fast breadth-first search implementation for Graph500. Technical Report UCB/EECS-2011-117, EECS Department, University of California, Berkeley, 2011.

[16] Scott Beamer, Krste Asanović, and David A. Patterson. Direction-optimizing breadth-first search. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[17] Scott Beamer, Krste Asanović, and David A. Patterson. GAIL: The graph algorithm iron law. *Workshop on Irregular Applications: Architectures and Algorithms (IA^3), at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[18] Scott Beamer, Krste Asanović, and David A. Patterson. The GAP benchmark suite. *arXiv*, 1508.03619 [cs.DC], Aug 2015.

[19] Scott Beamer, Krste Asanović, and David A. Patterson. Locality exists in graph processing: Workload characterization on an Ivy Bridge server. *International Symposium on Workload Characterization (IISWC)*, 2015.

[20] Scott Beamer, Aydın Buluç, Krste Asanović, and David A. Patterson. Distributed memory breadth-first search revisited: Enabling bottom-up search. *Workshop on Multithreaded Architectures and Applications (MTAAP), at the International Parallel & Distributed Processing Symposium (IPDPS)*, 2013.

[21] Michael A Bender, Gerth Stølting Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 61–70, 2007.

[22] Pavel Berkhin. A survey on pagerank computing. *Internet Mathematics*, 2(1):73–120, 2005.

[23] Jonathan Berry, Bruce Hendrickson, Simon Kahan, and Petr Konecny. Software and algorithms for graph queries on multithreaded architectures. *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2007.

[24] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing (JPDC)*, 37(1):55–69, 1996.

[25] Ronald F Boisvert, Roldan Pozo, and Karin A Remington. The matrix market exchange formats: Initial design. *National Institute of Standards and Technology Internal Report, NISTIR*, 5935, 1996.

[26] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. UbiCrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[27] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. *International World Wide Web Conference (WWW)*, 2011.

[28] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. *International World Wide Web Conference (WWW)*, pages 595–601, 2004.

[29] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[30] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 233–244, 2009.

[31] A. Buluç and J.R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications (IJHPCA)*, 25(4):496–509, 2011.

[32] Aydın Buluç, Scott Beamer, Kamesh Madduri, Krste Asanović, and David Patterson. Distributed-memory breadth-first search on massive graphs. In D. Bader, editor, *Parallel Graph Algorithms*. CRC Press, Taylor-Francis, 2017 (in press).

[33] Aydın Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[34] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on GPUs. *IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151, 2012.

[35] Umit V Catalyurek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[36] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. *SIAM Data Mining*, 2004.

[37] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. Pannotia: Understanding irregular GPGPU graph applications. *IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

[38] Fabio Checconi and Fabrizio Petrini. Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines. *International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 425–434, 2014.

[39] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. *European Conference on Computer Systems (EuroSys)*, pages 85–98, 2012.

[40] Jatin Chhugani, Nadathur Satish, Changkyu Kim, Jason Sewall, and Pradeep Dubey. Fast and efficient graph traversal algorithm for CPUs: Maximizing single-node efficiency. *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2012.

[41] Jee Whan Choi, Daniel Bedard, Robert Fowler, and Richard Vuduc. A roofline model of energy. *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2013.

[42] Shumo Chu and James Cheng. Triangle listing in massive networks and its applications. *Conference on Knowledge Discovery and Data Mining (KDD)*, Jun 2011.

[43] Guojing Cong and Konstantin Makarychev. Optimizing large-scale graph analysis on multithreaded, multicore platforms. *International Symposium on Parallel & Distributed Processing (IPDPS)*, Feb 2011.

[44] Guojing Cong and Simone Sbaraglia. A study on the locality behavior of minimum spanning tree algorithms. In *High Performance Computing (HiPC)*, pages 583–594. Springer, 2006.

[45] Convey HC-1 family. `www.conveycomputer.com/Resources/Convey_HC1_Family.pdf`, 2011.

[46] Elizabeth Cuthill and James McKee. Reducing the bandwidth of sparse symmetric matrices. *ACM National Conference*, pages 157–172, 1969.

[47] Timothy Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38:1:1 – 1:25, 2011.

[48] Michael deLorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomas E Uribe, Thomas Jr. Knight, and Andre DeHon. GraphStep: A system architecture for sparse-graph algorithms. *Symposium on Field-Programmable Custom Computing Machines*, pages 143–151, 2006.

[49] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[50] 9th DIMACS implementation challenge - shortest paths. `www.dis.uniroma1.it/challenge9`, 2006.

[51] Niels Doekemeijer and Ana Lucia Varbanescu. A survey of parallel graph processing frameworks. Technical Report PDS-2014-003, Delft University of Technology, 2014.

[52] Jack Dongarra and Piotr Luszczek. Introduction to the HPC challenge benchmark suite. Technical Report ICL-UT-05-01, Innovative Computing Laboratory (ICL), University of Tennessee, 2004.

[53] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. *International Conference on Supercomputing (ICS)*, pages 68–75, 1997.

[54] David Ediger, Karl Jiang, Jason Riedy, and David A Bader. Massive streaming data analytics: A case study with clustering coefficients. *Workshop on Multithreaded Architectures and Applications (MTAAP), at the International Parallel & Distributed Processing Symposium (IPDPS)*, pages 1–8, 2010.

[55] David Ediger, Rob McColl, Jason Riedy, and David A Bader. STINGER: High performance data structure for streaming graphs. *Conference on High Performance Extreme Computing (HPEC)*, pages 1–5, 2012.

[56] Susan J Eggers, Joel S Emer, Henry M Leby, Jack L Lo, Rebecca L Stamm, and Dean M Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997.

[57] Joel Emer and Douglas Clark. A characterization of processor performance in the VAX-11/780. *International Symposium on Computer Architecture (ISCA)*, 1984.

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

[59] Katsuki Fujisawa, Koji Ueno, Hitoshi Sato, Yuichiro Yasui, Keita Iwabuchi, Toyotaro Suzumura, and Ryo Mizote. A challenge to Graph500 benchmark: Trillion-scale graph processing on K computer. *International Supercomputing Conference (ISC): HPC in Asia*, 2015.

[60] GAP benchmark suite reference code v0.9. `github.com/sbeamer/gapbs`, 2016.

[61] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 345–354, 2012.

[62] Norman E Gibbs, William G Poole Jr, and Paul K Stockmeyer. An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM Journal on Numerical Analysis*, 13(2):236–250, 1976.

[63] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.

[64] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 599–613, 2014.

[65] Google C++ style guide. `google.github.io/styleguide/cppguide.html`, 2016.

[66] Graph500 benchmark. `www.graph500.org`, 2010.

[67] Graph500 SSSP proposal. `www.cc.gatech.edu/~jriedy/tmp/graph500`, 2010.

[68] Oded Green, Marat Dukhan, and Richard Vuduc. Branch-avoiding graph algorithms. *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015.

[69] Douglas Gregor and Andrew Lumsdaine. The parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, page 2, 2005.

[70] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single pc. *International Conference on Knowledge Discovery and Data Mining (KDD)*, 2013.

[71] Harshvardhan, A Fidel, N Amato, and L Rauchwerger. The STAPL parallel graph library. *Languages and Compilers for Parallel Computing*, 7760:46–60, Jan 2013.

[72] Sébastien Hily and André Seznec. Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading. *International Symposium on High-Performance Computer Architecture (HPCA)*, 1999.

[73] Brandon Holt, Jacob Nelson, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Flat combining synchronized global data structures. *International Conference on PGAS Programming Models*, 7, 2013.

[74] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. Green-Marl: A dsl for easy and efficient graph analysis. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[75] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. PGX.D: A fast distributed graph processing engine. *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[76] Sungpack Hong, Jan Van Der Lugt, Adam Welc, Raghavan Raman, and Hassan Chafi. Early experiences in using a domain-specific language for large-scale graph analysis. *Workshop on Graph Data Management Experience and Systems (GRADES)*, 2013.

[77] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. *Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[78] Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. Simplifying scalable graph processing with a domain-specific language. *International Symposium on Code Generation and Optimization (CGO)*, page 208, 2014.

[79] Imranul Hoque and Indranil Gupta. LFGraph: Simple and fast distributed graph analytics. *Conference on Timely Results in Operating Systems (TRIOS)*, Jan 2013.

[80] Intel 64 and IA-32 architectures optimization reference manual. September 2015.

[81] Intel performance counter monitor. `www.intel.com/software/pcm`, 2013.

[82] Ralph Johnson, Erich Gamma, Richard Helm, and John Vlissides. Design patterns: Elements of reusable object-oriented software. *Boston, Massachusetts: Addison-Wesley*, 1995.

[83] Shoaib Kamil. Asp: A SEJITS implementation for python. `github.com/shoaibkamil/asp`, 2011.

[84] Shoaib Kamil, Derrick Coetzee, Scott Beamer, Henry Cook, Ekaterina Gonina, Jonathan Harper, Jeffrey Morlan, and Armando Fox. Portable parallel performance from sequential, productive, embedded domain-specific languages. *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 47(8):303–304, 2012.

[85] U Kang, Charalampos Tsourakakis, and Christos Faloutsos. Pegasus: mining peta-scale graphs. *Knowledge and information systems*, 27(2):303–325, 2011.

[86] George Karypis and Vipin Kumar. METIS–unstructured graph partitioning and sparse matrix ordering system. version 2.0. `www.cs.umn.edu/~metis`, 1995.

[87] Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*. SIAM, 2011.

[88] Z Khayyat, K Awara, A Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. *European Conference on Computer Systems (EuroSys)*, 2013.

[89] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. CuSha: vertex-centric graph processing on gpus. *International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, pages 239–252, 2014.

[90] Christian Kohlschütter, Paul-Alexandru Chirita, and Wolfgang Nejdl. Efficient parallel computation of pagerank. *European Conference on Information Retrieval*, pages 241–252, 2006.

[91] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? *International World Wide Web Conference (WWW)*, 2010.

[92] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–17, Oct 2012.

[93] Michael M Lee, Indrajit Roy, Alvin AuYoung, Vanish Talwar, KR Jayaram, and Yuanyuan Zhou. Views and transactional storage for large graphs. In *Middleware*, pages 287–306. Springer, 2013.

[94] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. *European Conference on Principles and Practice of Knowledge Discovery in Databases*, 2005.

[95] Hang Liu and Howie Huang. Enterprise: breadth-first graph traversal on GPUs. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[96] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A new framework for parallel machine learning. *Uncertainty in Artificial Intelligence*, 2010.

[97] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Carlos Guestrin Danny Bickson, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *VLDB*, 2012.

[98] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[99] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. LLAMA: Efficient graph analytics using large multiversioned arrays. *International Conference on Data Engineering*, pages 363–374, 2015.

[100] Kamesh Madduri, David A Bader, Jonathan W Berry, and Joseph R Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. *ALENEX*, 7:23–35, 2007.

[101] Kamesh Madduri, David Ediger, Karl Jiang, David A Bader, and Daniel Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009.

[102] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. *International Conference on Management of Data (SIGMOD)*, Jun 2010.

[103] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A Bader. A performance evaluation of open source graph databases. *Workshop on Parallel Programming for Analytics Applications*, pages 11–18, 2014.

[104] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.

[105] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what COST? *Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[106] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. *Principles and Practice of Parallel Programming*, 2012.

[107] Ulrich Meyer and Peter Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[108] Scott Meyers. *Effective C++: 55 specific ways to improve your programs and designs.* Pearson Education, 2005.

[109] Stanley Milgram. The small world problem. *Psychology Today*, 2(1):60–67, 1967.

[110] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. *Conference on Internet Measurement (IMC)*, 2007.

[111] David Mizell and Kristyn Maschhoff. Early experiences with large-scale Cray XMT systems. *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009.

[112] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. *Department of Defense HPCMP Users Group Conference*, 1999.

[113] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Whirlpool: Improving dynamic cache management with static data classification. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 113–127, 2016.

[114] Richard C Murphy and Peter M Kogge. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *IEEE Transactions on Computers*, 56(7):937–945, 2007.

[115] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 129–140, 2003.

[116] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. GraphBIG: Understanding graph computing in the context of industrial solutions. *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[117] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[118] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Grappa: A latency-tolerant runtime for large-scale irregular applications. Technical report, University of Washington, 2014.

[119] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. *USENIX Annual Technical Conference*, pages 291–305, 2015.

[120] Jacob Nelson, Brandon Myers, A. H. Hunter, Preston Briggs, Luis Ceze, Carl Ebeling, Dan Grossman, Simon Kahan, and Mark Oskin. Crunching large graphs with commodity processors. *USENIX Conference on Hot Topics in Parallelism (HotPAR)*, 2011.

[121] Mario Nemirovsky and Dean M. Tulsen. *Multithreading Architecture*. Morgan & Claypool Publishers, 2012.

[122] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. *Symposium on Operating Systems Principles (SOSP)*, 2013.

[123] Rajesh Nishtala, Richard W Vuduc, James W Demmel, and Katherine A Yelick. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):297–311, 2007.

[124] Tayo Oguntebi and Kunle Olukotun. GraphOps: A dataflow library for graph analytics acceleration. *International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 111–117, 2016.

[125] Molly A O'Neil and Martin Burtscher. Microarchitectural performance characterization of irregular GPU kernels. *IEEE International Symposium on Workload Characterization (IISWC)*, 2014.

[126] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. Energy efficient architecture for graph analytics accelerators. *International Symposium on Computer Architecture (ISCA)*, 2016.

[127] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[128] Thap Panitanarak and Kamesh Madduri. Performance analysis of single-source shortest path algorithms on distributed-memory systems. *SIAM Workshop on Combinatorial Scientific Computing (CSC)*, page 60, 2014.

[129] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 4th edition, 2008.

[130] Roger Pearce, Maya Gokhale, and Nancy M Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[131] Tiago P. Peixoto. The graph-tool python library. `graph-tool.skewed.de`, 2014.

[132] Jose B Pereira-Leal, Anton J Enright, and Christos A Ouzounis. Detection of functional modules from protein interaction networks. *PROTEINS: Structure, Function, and Bioinformatics*, 54(1):49–57, 2004.

[133] Orestis Polychroniou and Kenneth A Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison-and radix-sort. *International Conference on Management of Data (SIGMOD)*, pages 755–766, 2014.

[134] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. Managing large graphs on multi-cores with graph awareness. *USENIX Annual Technical Conference*, 12, 2012.

[135] Steven E Raasch and Steven K Reinhardt. The impact of resource partitioning on SMT processors. *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 15–25, 2003.

[136] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. *Symposium on Operating Systems Principles (SOSP)*, pages 410–424, 2015.

[137] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: edge-centric graph processing using streaming partitions. *Symposium on Operating Systems Principles (SOSP)*, 2013.

[138] Semih Salihoglu and Jennifer Widom. GPS: A graph processing system. International Conference on Scientific and Statistical Database Management:1–31, 2013.

[139] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on Pregel-like systems. Technical report, Stanford University, 2014.

[140] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. On the surprising difficulty of simple things: the case of radix partitioning. *VLDB*, 8(9):934–937, 2015.

[141] Daniel A Schult and P Swart. Exploring network structure, dynamics, and function using NetworkX. *Python in Science Conferences (SciPy)*, 2008:11–16, 2008.

[142] J Seo, J Park, J Shin, and M Lam. Distributed SociaLite: A Datalog-based language for large-scale graph analysis. *VLDB*, Jan 2013.

[143] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. *International Conference on Management of Data (SIGMOD)*, pages 505–516, 2013.

[144] Yossi Shiloach and Uzi Vishkin. An $o(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[145] Julian Shun. *Shared-Memory Parallelism Can Be Simple, Fast, and Scalable*. PhD thesis, Carnegie Mellon University (CMU), 2015.

[146] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 48(8):135–146, 2013.

[147] Julian Shun, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 68–70, 2012.

[148] Jeremy G Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Pearson Education, 2001.

[149] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, and Kuansan Wang. An overview of Microsoft academic service (MAS) and applications. *World Wide Web Consortium (W3C)*, 2015.

[150] Burton J Smith. A pipelined, shared resource MIMD computer. *Advanced Computer Architecture*, pages 39–41, 1986.

[151] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. *International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1222–1230, 2012.

[152] Philip Stutz, Abraham Bernstein, and William Cohen. Signal/collect: graph algorithms for the (semantic) web. *International Semantic Web Conference (ISWC)*, pages 764–780, 2010.

[153] Narayanan Sundaram, Nadathur Rajagopalan Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. GraphMat: High performance graph analytics made productive. *VLDB*, 2015.

[154] Keith D Underwood, Megan Vance, Jonathan Berry, and Bruce Hendrickson. Analyzing the scalability of graph algorithms on Eldorado. *International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–8, 2007.

[155] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM (CACM)*, 33(8):103–111, 1990.

[156] Kenzo Van Craeynest, Stijn Eyerman, and Lieven Eeckhout. MLP-aware runahead threads in a simultaneous multithreading processor. *International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC)*, pages 110–124, 2008.

[157] Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert Schreiber. Presto: distributed machine learning and graph processing with sparse matrices. *European Conference on Computer Systems (EuroSys)*, pages 197–210, 2013.

[158] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the GPU. *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[159] Jan Wassenberg and Peter Sanders. Engineering a multi-core radix sort. In *Euro-Par Parallel Processing*, pages 160–169. Springer, 2011.

[160] Duncan Watts and Steven Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, June 1998.

[161] Wikipedia page-to-page link database. `haselgrove.id.au/wikipedia.htm`, 2009.

[162] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[163] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P.N. Puttaswamy, and Ben Y. Zhao. User interactions in social networks and their implications. *European Conference on Computer Systems (EuroSys)*, 2009.

[164] Yuduo Wu, Yangzihao Wang, Yuechao Pan, Carl Yang, and John D. Owens. Performance characterization for high-level programming models for GPU graph analytics. *International Symposium on Workload Characterization (IISWC)*, 2015.

[165] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. GraphX: A resilient distributed graph system on spark. *International Workshop on Graph Data Management Experiences and Systems (GRADES)*, 2013.

[166] Qiumin Xu, Hyeran Jeon, and Murali Annavaram. Graph processing on GPUs: Where are the bottlenecks? *IEEE International Symposium on Workload Characterization (IISWC)*, 2014.

[167] Jilong Xue, Zhi Yang, Zhi Qu, Shian Hou, and Yafei Dai. Seraph: an efficient, low-cost system for concurrent graph processing. *International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, pages 227–238, 2014.

[168] J Yan, G Tan, and N Sun. GRE: A graph runtime engine for large-scale distributed graph-parallel applications. *arXiv*, 1310.5603 [cs.DC], Jan 2013.

[169] Jie Yan, Guangming Tan, Zeyao Mo, and Ninghui Sun. Graphine: Programming graph-parallel computation of large natural graphs for multicore clusters. *Transactions on Parallel and Distributed Systems*, 27(6):1647–1659, 2016.

[170] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *arXiv*, 1205.6233, 2012.

[171] Shengqi Yang, Xifeng Yan, Bo Zong, and Arijit Khan. Towards effective partition management for large graphs. *International Conference on Management of Data (SIGMOD)*, pages 517–528, 2012.

[172] Jin Y Yen. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quarterly of Applied Mathematics*, 27(4):526–530, 1970.

[173] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Ümit Çatalyürek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2005.

[174] Kisun You, Jike Chong, Youngmin Yi, Ekaterina Gonina, Christopher Hughes, Yen-Kuang Chen, Wonyong Sung, and Kurt Keutzer. Scalable HMM-based inference engine in large vocabulary continuous speech recognition. *IEEE Signal Processing Magazine*, 2010.

[175] Liang Yuan, Chen Ding, D Tefankovic, and Yunquan Zhang. Modeling the locality in graph traversals. *International Conference on Parallel Processing (ICPP)*, 2012.

[176] Albert-Jan N Yzelman and Rob H Bisseling. A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve. In *Progress in Industrial Mathematics at ECMI*, pages 627–633. Springer, 2012.

[177] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. FlashGraph: Processing billion-node graphs on an array of commodity ssds. *Conference on File and Storage Technologies (FAST)*, pages 45–58, 2015.

[178] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on GPUs. *Transactions on Parallel and Distributed Systems (TPDS)*, 25(6):1543–1552, 2014.

[179] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. *USENIX Annual Technical Conference*, pages 375–386, 2015.

Appendix A

Most Commonly Evaluated Graph Kernels

To guide the selection of evaluation kernels for our work, we surveyed the literature of graph processing frameworks. We selected the six most commonly evaluated kernels, and Table A.1 lists the frameworks and shows which graph kernels the frameworks use to evaluate themselves. Due to the generality of the graph abstraction, these graph processing publications are published in a variety of areas including: high-performance computing, theory, databases, and computer architecture. Additionally, the frameworks target a variety of hardware platforms including shared memory, distributed memory (clusters), semi-external memory (flash/SSD), or even custom hardware accelerators (FPGA/ASIC). In Table A.1 we show only the 10 most commonly evaluated kernels, but there is a long tail on the distribution of evaluated kernels, as we observed at least 64 different kernels. Some frameworks appear more than once as we track their development over time. Additionally, some frameworks support multiple classes of hardware platform, but in our table we categorize them by their most distinguishing platform.

For more information on the breadth of graph processing frameworks, we recommend a few surveys. Doekemeijer et al. provide a great overview, as they consider multiple hardware platforms and they provide a taxonomy of programming models and framework features [51]. McCune et al. focus on graph frameworks that use the "think like a vertex model" such as Pregel [102, 104]. McColl et al. survey open-source graph databases and include performance comparisons and commentary on usability [103].

Framework	Year	Platform	Total	PageRank	Single-source Shortest Paths	Connected Components	Breadth-first Search	Triangle Counting	Betweenness Centrality	Conductance	n-Hop Queries	Alternating Least Squares	Strongly Connected Components	Other
MTGL [23]	2007	Shared Memory	3			✓								2
SNAP [10]	2008		3											3
GraphLab [96]	2010		5											5
Signal/Collect [152]	2010		5	✓	✓									3
Grace [134]	2012		6	✓	✓	✓	✓				✓			1
Green-Marl [74]	2012		5	✓					✓	✓			✓	1
Galois [122]	2013		6	✓	✓	✓	✓		✓					1
Ligra [146]	2013		6	✓	✓	✓	✓		✓					1
STINGER [103]	2014		3	✓	✓	✓								0
EmptyHeaded [1]	2015		3	✓	✓			✓						0
GraphMat [153]	2015		5	✓	✓		✓	✓						1
Pearce et al. [130]	2010	Flash/SSD	3		✓	✓	✓							0
GraphChi [92]	2012		6	✓		✓		✓				✓		2
TurboGraph [70]	2013		2				✓				✓			0
X-Stream [137]	2013		11	✓	✓	✓	✓			✓		✓	✓	4
FlashGraph [177]	2015		6	✓		✓	✓	✓	✓					1
GridGraph [179]	2015		4	✓		✓	✓							1
LLAMA [99]	2015		3	✓			✓	✓						0
Totem [61]	2012	GPU	2	✓			✓							0
CuSha [89]	2014		8	✓	✓	✓	✓							4
Medusa [178]	2014		4	✓	✓		✓							1
Gunrock [158]	2016		5	✓	✓	✓	✓		✓					0
Tesseract [4]	2015	Acc.	5	✓	✓					✓				2
GraphOps [124]	2016		6	✓	✓		✓			✓				2
Ozdal et al. [126]	2016		4	✓	✓									2

Table A.1: Graph kernels used in framework evaluations

Framework	Year	Platform	Total	PageRank	Single-source Shortest Paths	Connected Components	Breadth-first Search	Triangle Counting	Betweenness Centrality	Conductance	n-Hop Queries	Alternating Least Squares	Strongly Connected Components	Other
PBGL [69]	2005	Distributed Memory	4		✓	✓								2
Pregel [102]	2010		4	✓	✓									2
CombBLAS [31]	2011		2						✓					1
Pegasus [85]	2011		4	✓					✓					2
D. Graphlab [97]	2012		3									✓		2
Kineograph [39]	2012		3											3
Powergraph [63]	2012		3	✓				✓						1
Sedge [171]	2012		3								✓			2
Concerto [93]	2013		5								✓			4
D. SociaLite [142]	2013		6	✓	✓	✓		✓						2
GPS [138]	2013		4	✓	✓	✓								1
GraphX [165]	2013		2	✓		✓								0
Grappa [73]	2013		2			✓	✓							0
GRE [168]	2013		3	✓	✓	✓								0
Green-Marl [76]	2013		3	✓				✓						1
LFGraph [79]	2013		3	✓	✓			✓						0
Mizan [88]	2013		2	✓										1
Presto [157]	2013		6	✓	✓			✓	✓			✓		1
STAPL [71]	2013		3	✓			✓							1
Trinity [143]	2013		4	✓			✓				✓			1
GPS [139]	2014		5			✓							✓	3
GraphX [64]	2014		2	✓		✓								0
Grappa [118]	2014		4	✓			✓							2
Green-Marl [78]	2014		6	✓	✓				✓	✓				2
Seraph [167]	2014		5	✓	✓	✓								2
Chaos [136]	2015		10	✓	✓	✓	✓			✓			✓	4
Grappa [119]	2015		4	✓	✓	✓	✓							0
PGX.D [75]	2015		6	✓	✓	✓	✓							2
Graphine [169]	2016		3	✓	✓	✓								0

Appendix B

GBSP: Graph Bulk Synchronous Parallel

In this chapter, we introduce a domain-specific language for graph processing that is both productive and high-performance. Its creation [84] preceded the work in the rest of this dissertation, but we include this appendix as its creation was the context in which our propagation blocking technique (Chapter 7) was conceived and first applied.

Introduction

Creating an application that is useful to the world often requires more than simply knowing how to program, as it often requires knowing about the application's domain. Application *domain experts* possess a great deal of knowledge about their application area, and they program in a manner that is most productive for their domain (e.g. MATLAB). By programming with a domain-specific language (DSL), domain experts are able to focus on the key parts of the application while worrying less about efficiency and implementation details. In addition to defining a language for a domain, creating a DSL requires substantial developer effort to implement an interpreter or an optimizing compiler. To make the DSL high-performance, the implementation effort also needs the assistance of an *efficiency programmer* who understands the performance challenges of the hardware platform.

In this work, we use the *Selective Embedded Just-In-Time Specialization (SEJITS)* approach, which reduces the burden of creating a productive high-performance DSL [84]. By embedding the DSL in an existing host language, the SEJITS approach is able to reuse the infrastructure of the host language. To improve performance, a *specializer* takes the DSL code and translates it into optimized code in an efficiency language (e.g., C++) in order to leverage existing optimizing compilers. To make this source-to-source translation tractable, the DSL should use domain-specific abstractions to keep the DSL simple and restrict the scope of legal programs.

Since the DSL is embedded in a host language, programs in the host language can easily include fragments of DSL code. These fragments are annotated such that when the host

language executes them, they invoke the DSL's specializer. The specializer uses the host language's reflection capabilities to examine the DSL fragment and decide if it can be specialized. If the fragment can be specialized, the specializer proceeds with the source-to-source translation. Once the source translation is complete, the optimized code is compiled, and the resulting binary is linked back into the current execution. By performing this translation at runtime, the specializer can consider attributes such as the input and the execution platform when performing its optimizations. If the fragment cannot be specialized, the DSL fragment can still be executed in the host language as a fallback since the DSL is embedded.

To ease the creation of specializers, the SEJITS infrastructure provides nearly all of the reusable functionality needed, allowing the specializer developer to spend more of their time on the domain-specific optimizations. The SEJITS infrastructure provides mechanisms to trigger the specializer as well as link the compiled native binary back in. It also provides utilities to ease writing the transformation passes. Additionally, the infrastructure can also cache the output of the specializer, so if the DSL input does not change, it can reuse the compiled binary.

For many domains, commonly used application kernels can easily be packaged and reused as library functions. The SEJITS approach is most useful when such approaches are insufficient, typically because the usage scenario requires more customizability or flexibility than is possible with a library function. This can occur if the application-specific logic is within a kernel rather than the composition of kernels. For example, a graph application may want to apply custom application logic at each vertex while reusing a library BFS traversal. Another advantage of specializing code fragments instead of using library function invocations is that the operations can be fused together to improve efficiency.

Graph Bulk Synchronous Parallel (GBSP)

We create the *Graph Bulk Synchronous Parallel (GBSP)* framework using the SEJITS approach to enable productive programming of high-performance graph algorithm implementations. For our GBSP DSL, we select the vertex-centric programming model which has already been proven successful by Pregel [102] and GraphStep [48]. In the vertex-centric model, each vertex is a parallel entity in a bulk synchronous parallel (BSP) [155] paradigm, so all interactions between vertices are performed by message passing. The computation is divided into global iterations that are synchronized by barriers. Within each iteration, a vertex can examine messages sent to it in the previous iteration, mutate its own local data, and send messages to its neighbors. At the end of the iteration, all vertices wait at a barrier as the messages are delivered. Performing a graph algorithm in this message passing style of communication is equivalent to performing a graph algorithm in the push direction.

To implement GBSP, we utilize the Asp (Asp is SEJITS in Python) framework [83]. Due to Python's popularity within the graph analysis domain [141], we select Python as our host language to ease adoption so that GBSP can be used alongside existing code. Using Asp, our specializer translates GBSP code snippets to C++ at runtime. The messaging phase of GBSP potentially has poor locality due to its scattered writes across inboxes, so we

Graph Kernel	Graph	Vertices (K)	Edges (K)	Directed
BFS	Kronecker [94]	1,048.6	15,763.2	
SSSP	`webbase-1m` [47]	1,000.0	3,105.5	✓
TC	Kronecker [94]	16.4	212.9	

Table B.1: Workload (graph kernels with their input graphs) for evaluation

use propagation blocking (Chapter 7) to improve locality. This application of propagation blocking is quite natural, as the propagations are explicit messages. The destination vertex values are inboxes, so the accumulate phase inserts messages into their destination inboxes. Since we use BSP as our underlying computation model, we are able to trivially parallelize GBSP, and we do so with OpenMP.

Evaluation

To evaluate GBSP, we compare it against three other graph frameworks:

Boost Graph Library (BGL) is a templated C++ library for graph algorithms [148]. Users of BGL can access implementations of existing kernels as well as specialize those kernels for their needs via the visitor pattern. BGL is single-threaded, and PBGL [69] is its parallelized successor, but we do not evaluate PBGL as it is designed for distributed memory. Since BGL is well implemented in C++, it is representative of native performance.

NetworkX is a Python framework intended for graph analysis [141]. It provides many great abstractions, but since it is implemented in Python, its performance is bottlenecked by the Python interpreter.

graph-tool attempts to combine the performance of BGL with the expressiveness of Python [131]. It allows its users to use BGL while expressing their visitor operations in Python. Internally, graph-tool makes extensive use of Python's foreign function interface (FFI) in order to inject its Python snippets into a C++ program using BGL.

To evaluate the frameworks, we use the small graph analysis workload shown in Table B.1. The graph sizes are small in order to accomodate the limitations of productivity language frameworks such as NetworkX.

We perform our evaluation on a dual-socket Intel Westmere (X5680) which has 12 hyper-threaded 3.33 GHz cores and 48 GB of DRAM. We first evaluate GBSP's serial performance since the frameworks we compare it against are all serial. Figure B.1 shows that GBSP is successful in obtaining native performance, since it is comparable to BGL and orders of magnitude faster than the productivity language framework NetworkX. The performance of graph-tool is more comparable to NetworkX than BGL since the time spent in Python dominates the execution time. GBSP, unlike the other graph frameworks we evaluate, can

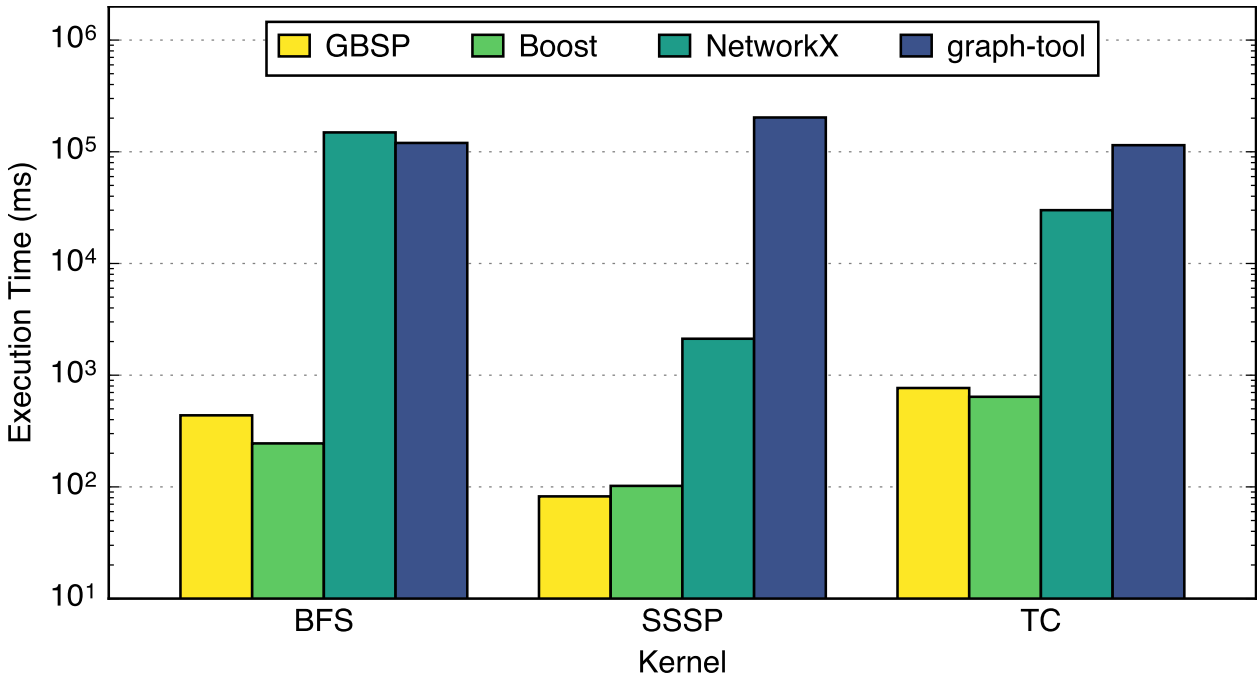

Figure B.1: Single-threaded performance comparison

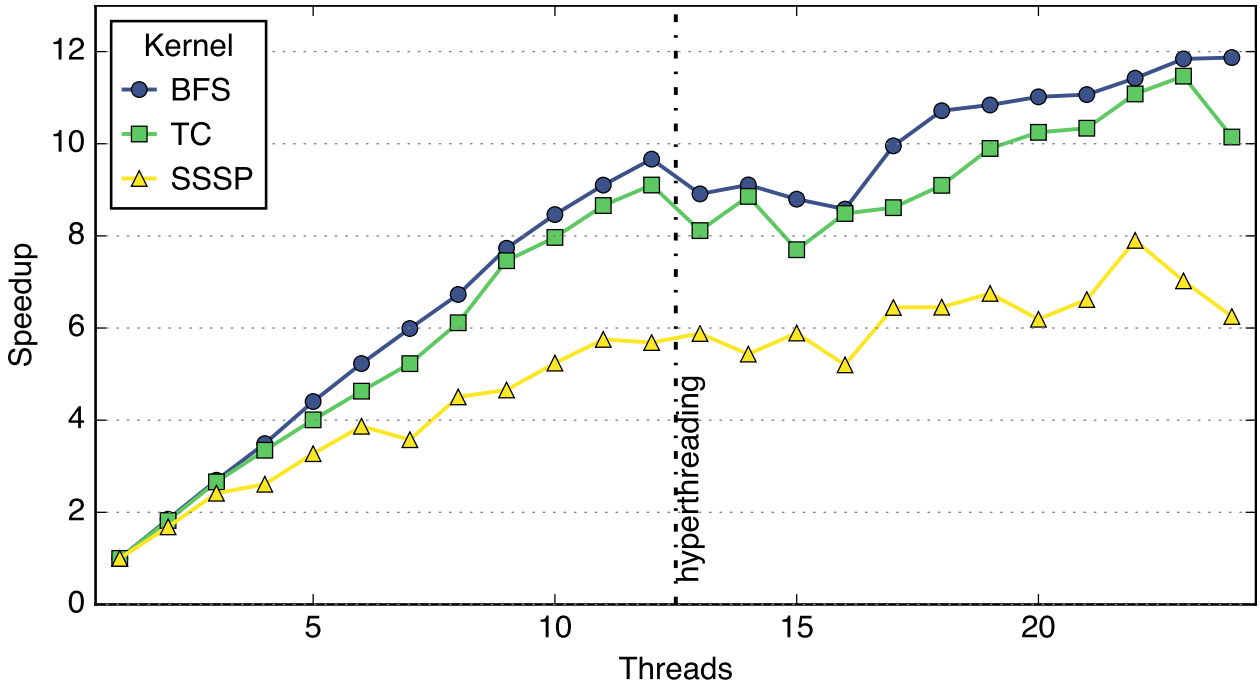

Figure B.2: GBSP parallel strong-scaling speedup relative to single-threaded performance

execute in parallel, and doing so results in modest speedups until hyperthreads are used (Figure B.2). These parallel speedups are a great result, as they require no additional effort by the GBSP user. GBSP is able to automatically transform the simple expressions of the algorithms into high-performance parallel code.

Conclusion

Achieving productivity and performance simultaneously is challenging, so GBSP makes two essential simplifications to the problem. First, rather than converting any Python code into high-performance code, it restricts itself to a simple well-defined DSL (graph algorithms expressed as BSP). Second, the specializer only needs to translate GBSP code to C++, so it can leverage existing optimizing compilers. Overall, GBSP demonstrates that the SEJITS methodology can be successful.