

ANALYZING HYBRID ARCHITECTURES FOR MASSIVELY PARALLEL GRAPH ANALYSIS

A Dissertation
Presented to
The Academic Faculty

by

David Ediger

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2013

Copyright © 2013 by David Ediger

ANALYZING HYBRID ARCHITECTURES FOR MASSIVELY PARALLEL GRAPH ANALYSIS

Approved by:

Dr. George Riley, Committee Chair
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Dr. David A. Bader, Advisor
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Dr. Bo Hong
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Dr. Linda Wills
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Dr. Rich Vuduc
School of Computational Science and
Engineering
Georgia Institute of Technology

Date Approved: 25 March 2013

*To my wife, Michelle,
and my parents, Mark and Merrilee,
for encouraging a passion for learning.*

ACKNOWLEDGEMENTS

I want to thank my friends and family that believed in me from the day I decided to go to graduate school. I want to make special mention of my fellow graduate students, past and present, for walking the road with me and sharing so much of themselves.

Thank you to my high performance computing colleagues, especially Rob McColl, Oded Green, Jason Riedy, Henning Meyerhenke, Seunghwa Kang, Aparna Chandramowlishwaran, and Karl Jiang. I will miss solving the world's most pressing problems with all of you.

Thank you to my advisor, David Bader, for introducing me to the world of graphs, and for inspiring and challenging me to complete my doctoral degree. Your encouragement and wisdom has helped me grow immensely as a researcher.

I want to thank the members of my committee: Professor Rich Vuduc, Professor Linda Wills, Professor Bo Hong, and Professor George Riley, who chaired the committee.

This work was supported in part by the Pacific Northwest National Lab (PNNL) Center for Adaptive Supercomputing Software for MultiThreaded Architectures (CASS-MT). I want to acknowledge the research staff at PNNL, Sandia National Laboratories, and Cray for their mentorship throughout my graduate studies.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGEMENTS	iv
LIST OF TABLES	viii
LIST OF FIGURES	ix
SUMMARY	xii
I INTRODUCTION	1
II ORIGIN AND HISTORY OF THE PROBLEM	5
2.1 Relevant Applications	6
2.1.1 Computational Biology	6
2.1.2 Business Analytics	7
2.1.3 Security	8
2.1.4 Social Networks	9
2.2 Architectures and Programming Models	10
2.2.1 Cray XMT	11
2.2.2 MapReduce	12
2.3 High Performance Graph Research	14
2.3.1 Parallel Boost Graph Library	15
2.3.2 GraphCT	15
2.3.3 Sandia Multithreaded Graph Library	16
2.3.4 Knowledge Discovery Toolbox	16
2.3.5 Google Pregel	17
III GRAPHCT	19
3.1 Connected Components	32
3.2 Clustering Coefficients	35
3.3 k -Betweenness Centrality	40

IV	STINGER	46
4.1	Experimental Setup	48
4.2	Optimizations	50
4.3	Streaming Clustering Coefficients	54
4.4	Streaming Connected Components	61
V	ALTERNATIVE PROGRAMMING MODELS	72
5.1	MapReduce	72
5.1.1	Data-parallelism and Locality	72
5.1.2	Load Balance	73
5.1.3	Resilience	74
5.1.4	Communication Costs	75
5.1.5	Complexity	76
5.1.6	Experimental Results	78
5.2	Bulk Synchronous Parallel and Pregel	81
5.2.1	Experimental Method	82
5.2.2	Connected Components	83
5.2.3	Breadth-first Search	87
5.2.4	Clustering Coefficients	92
5.2.5	Discussion	94
5.3	Random Access and Storage Performance	97
5.3.1	Measuring Data Access Time	97
5.3.2	Building a Model for Access Time	103
5.3.3	Estimating Concurrency versus I/O Time	110
VI	A HYBRID SYSTEM ARCHITECTURE	113
6.1	A Hierarchy of Data Analytics	113
6.2	Modeling a Hybrid System	117
6.2.1	Experimental Method	119
6.2.2	Results	120

6.2.3 Discussion	125
VII RISE OF THE MACROARCHITECTURE	130
7.1 Parallelism	131
7.2 Shared Memory and BSP	132
7.3 Latency-oriented Analytics	133
7.4 Memory Technologies	135
7.5 Reliability & Resilience	136
7.6 Integration	137
VIII CONCLUSION	140
REFERENCES	145
VITA	154

LIST OF TABLES

1	Graph analysis packages and frameworks currently under development.	21
2	Running times in seconds for connected components on a 128-processor Cray XMT.	34
3	The number of vertices ranked in selected percentiles for $k = 1$ and $k = 2$ whose betweenness centrality score was 0 for $k = 0$ (traditional BC). There were 14,320 vertices whose traditional BC score was 0, but whose BC_k score for $k = 1$ was greater than 0. The ND-www graph contains 325,729 vertices and 1,497,135 edges.	43
4	Summary of update algorithms	60
5	Comparison of single edge versus batched edge updates on 32 Cray XMT processors, in updates per second, on a scale-free graph with approximately 16 million vertices and 135 million edges.	61
6	Updates per second on a graph starting with 16 million vertices and approximately 135 million edges on 32 processors of a Cray XMT. . .	68
7	Execution times on a 128-processor Cray XMT for an undirected, scale-free graph with 16 million vertices and 268 million edges.	95
8	Number of data elements that can be read per second as determined by microbenchmarks.	102
9	Estimated execution of breadth-first search on a graph with 17.2 billion vertices and 137 billion edges.	106
10	Experimental parameters	119

LIST OF FIGURES

1	GraphCT is an open source framework for developing scalable multi-threaded graph analytics in a cross-platform environment.	20
2	An example user analysis workflow in which the graph is constructed, the vertices are labeled according to their connected components, and a single component is extracted for further analysis using several complex metrics, such as betweenness centrality.	21
3	This MTA pragma instructs the Cray XMT compiler that the loop iterations are independent.	24
4	Scalability of Shiloach-Vishkin connected components with and without tree-climbing optimization on the Cray XMT. The input graph is scale-free with approximately 2 billion vertices and 17 billion edges. The speedup is 113x on 128 processors.	33
5	There are two triplets around v in this unweighted, undirected graph. The triplet (m, v, n) is open, there is no edge $\langle m, n \rangle$. The triplet (i, v, j) is closed.	35
6	Scalability of the local clustering coefficients kernel on the Cray XMT. On the left, the input graph is an undirected, scale-free graph with approximately 16 million vertices and 135 million edges. The speedup is 94x on 128 processors. On the right, the input graph is the USA road network with 24 million vertices and 58 million edges. The speedup is 120x on 128 processors. Execution times in seconds are shown in blue.	37
7	Scalability of the transitivity coefficients kernel on the Cray XMT. The input graph is a directed, scale-free graph with approximately 16 million vertices and 135 million edges. Execution times in seconds are shown in blue. On 128 processors, we achieve a speedup of 90x.	38
8	Parallel scaling on the Cray XMT, for a scale-free, undirected graph with approximately 16 million vertices and 135 million edges. Scaling is nearly linear up to 96 processors and speedup is roughly 78 on all 128 processors. $k = 1$ with 256 random sources (single node time 318 minutes). Execution times in seconds shown in blue.	41
9	Per-vertex betweenness centrality scores for $k = 0, 1,$ and $2,$ sorted in ascending order for $k = 0.$ Note the number of vertices whose score is several orders of magnitude larger for $k = 1$ or 2 than for traditional betweenness centrality.	44
10	Per-vertex betweenness centrality scores for $k = 0, 1,$ and $2,$ sorted independently in ascending order for each value of $k.$	44

11	Updates per second on a scale-free graph with approximately 16 million vertices and 270 million edges and a batch size of 100,000 edge updates.	51
12	Increasing batch size results in better performance on the 128-processor Cray XMT. The initial graph is a scale-free graph with approximately 67 million vertices and 537 million edges.	51
13	There are two triplets around v in this unweighted, undirected graph. The triplet (m, v, n) is open, there is no edge $\langle m, n \rangle$. The triplet (i, v, j) is closed.	55
14	Speedup of incremental, local updates relative to recomputing over the entire graph.	59
15	Updates per second by algorithm.	60
16	Update performance for a synthetic, scale-free graph with 1 million vertices (left) and 16 million vertices (right) and edge factor 8 on 32 processors of a 128 processor Cray XMT.	70
17	Workflow of a single MapReduce job.	76
18	The number of reads and writes per iteration performed by connected components algorithms on a scale-free graph with 2 million vertices and 131 million edges.	85
19	Execution time per iteration performed by connected components algorithms. Scale is the log base 2 of the number of vertices and the edge factor is 8. For Scale 21, shared memory completes in 8.3 seconds, while BSP takes 12.7 seconds.	86
20	Connected components execution time by iteration for an undirected, scale-free graph with 16 million vertices and 268 million edges. On the 128-processor Cray XMT, BSP execution time is 5.40 seconds and GraphCT execution time is 1.31 seconds.	87
21	Size of the frontier as a function of breadth-first search level.	89
22	Scalability of breadth-first search levels 3 to 8 on an undirected, scale-free graph with 16 million vertices and 268 million edges. Total execution time on the 128-processor Cray XMT is 3.12 seconds for BSP and 310 milliseconds for GraphCT.	91
23	Scalability of triangle counting algorithms on an undirected, scale-free graph with 16 million vertices and 268 million edges. Execution time on the 128-processor Cray XMT is 444 seconds for BSP and 47.4 seconds for GraphCT.	95
24	Normalized execution time for an array memory access microbenchmark.	98

25	Normalized execution time comparing several static graph algorithms with linear and random array memory access.	100
26	Iso-performance contours (blue) varying concurrency of DRAM and SSD devices in the hybrid system. Red lines indicate percentage improvement over the in-memory graph algorithm with the same memory concurrency. The input graph has 1.1 trillion vertices and 17.6 trillion edges.	109
27	Estimated I/O time for graph algorithms using hard disk drives and solid state drives. Dashed line indicates in-memory computation time.	112
28	A proposed hierarchy of data analytics the includes raw, unstructured data in a commodity cluster, a special purpose high-performance graph representation, and many client analytics that operate in parallel. . .	115
29	Connected components execution time comparison for a hybrid system with a shared memory graph and SSD-based storage system. The input data is an undirected graph with 1.1 trillion vertices and 17.6 trillion edges.	121
30	Breadth-first search execution time comparison for a hybrid system with a shared memory graph and SSD-based storage system. The input data is an undirected graph with 1.1 trillion vertices and 17.6 trillion edges.	123
31	Triangle counting execution time comparison for a hybrid system with a shared memory graph and SSD-based storage system. The input data is an undirected graph with 4.3 billion vertices and 68.7 billion edges.	124
32	The total number of random and linear data references by algorithm. The input graph has 17.6 trillion edges.	126
33	Algorithm modeling random and linear access by quantity and time for hard disk drives and solid state disks. Breadth-first search and connected components are modeled for a graph with 1.1 trillion vertices and 17.6 trillion edges. Triangle counting is modeled for a graph with 4.3 billion vertices and 68.7 billion edges.	128

SUMMARY

The quantity of rich, semi-structured data generated by sensor networks, scientific simulation, business activity, and the Internet grows daily. The objective of this research is to investigate architectural requirements for emerging applications in massive graph analysis. Using emerging hybrid systems, we will map applications to architectures and close the loop between software and hardware design in this application space. Parallel algorithms and specialized machine architectures are necessary to handle the immense size and rate of change of today's graph data. To highlight the impact of this work, we describe a number of relevant application areas ranging from biology to business and cybersecurity. With several proposed architectures for massively parallel graph analysis, we investigate the interplay of hardware, algorithm, data, and programming model through real-world experiments and simulations. We demonstrate techniques for obtaining parallel scaling on multithreaded systems using graph algorithms that are orders of magnitude faster and larger than the state of the art. The outcome of this work is a proposed hybrid architecture for massive-scale analytics that leverages key aspects of data-parallel and highly multithreaded systems. In simulations, the hybrid systems incorporating a mix of multithreaded, shared memory systems and solid state disks performed up to twice as fast as either homogeneous system alone on graphs with as many as 18 trillion edges.

CHAPTER I

INTRODUCTION

The quantity of rich, semi-structured data generated by sensor networks, scientific simulation, business activity, and the Internet grows daily. In the past, collection and analysis of data using relational queries was sufficient. Today, complex analytics and near real-time responses to new data are required. To meet the demand and keep up with ever-increasing data rates, novel solutions in the form of hardware, software, and algorithms are required.

The objective of this research is to investigate architectural requirements for emerging applications in massive graph analysis. Using emerging hybrid systems, we will map applications to architectures and close the loop between software and hardware design in this application space.

A brief summary of graph theory is presented in Chapter 2. It is argued that parallel algorithms and specialized machine architectures are necessary to handle the immense size and rate of change of today's graph data. To highlight the impact of this work, we investigate a number of relevant application areas ranging from biology to business and cybersecurity. We will see that there are several competing architectures and programming models for massively parallel graph analysis today. We also present several research efforts in high performance graph analytics, including our work at Georgia Institute of Technology.

Chapter 3 presents GraphCT, a graph analysis toolkit that contains parallel graph algorithms and an in-memory workflow. We offer an in-depth study of several algorithms and their performance and scalability on the Cray XMT. Techniques for algorithm engineering and software development are described. GraphCT is an open

source package that compiles and runs on the Cray XMT and x86 workstations with OpenMP.

Turning from static to dynamic graph analysis, Chapter 4 examines data structures and new algorithms for online analysis of graph data in a streaming and semantic environment. STINGER is an open source, hybrid parallel data structure for dynamic, streaming graphs that contain rich semantic data and runs on the Cray XMT and x86 systems. We demonstrate techniques for obtaining high performance and scalability in STINGER. We describe several new algorithms for keeping analytics, such as clustering coefficients and connected components, up to date given an infinite stream of edge insertions and deletions.

In Chapter 5, we describe and consider several alternative programming models for graph analytics. Graph applications are often designed in the shared memory model, but new technologies, such as MapReduce clouds, can be considered as alternative styles. We show that MapReduce is a poor choice for most graph algorithms, but the bulk synchronous parallel (BSP) programming model is potentially useful for graphs. We study several algorithms in the BSP model and measure their performance and scalability on existing systems. To accommodate future data volumes, we construct a model that estimates algorithm performance given a combination of main memory, hard disk drives, and solid state disks. We use the model to estimate the concurrency required for each to meet a given performance target.

Chapter 6 describes a data analysis hierarchy of disk- and memory-based systems and argues that analytics can be run throughout the hierarchy wherever it is most efficient given the state of the data and the algorithm. We use the data access model from Chapter 5 to estimate algorithm performance in the the hybrid system and model under what conditions each system architecture should be used for best performance. We find that a small number of random data references greatly impacts execution time when the disparity between random and linear access rates is large.

For maximum impact on future system designs, Chapter 7 summarizes the important architecture-related experimental results and considers the challenges of future system design in several key areas related to data-intensive analytics. Applying the conclusions of this thesis, a discussion of each of the key recommendations for massive graph analysis follows. The computer architecture community is challenged to consider the role of macroarchitecture research in future system designs.

Across these chapters, we witness a compelling argument for broad data-driven research that combines new algorithms and programming models, runtime systems, hardware devices and system architectures, and analysis techniques. At the conclusion of this research, we more fully understand the algorithmic techniques necessary to increase parallelism and obtain strong scalability with tens of thousands of threads on scale-free graphs. We have created high performance data structures and algorithms to cope with high rates of change in network topologies. We have analyzed graph algorithms in light of the programming model and the hardware they are executed on, and modeled their performance on future hybrid systems. Future research in this area will continue to require co-analysis of algorithm, machine, and data for high performance.

This dissertation is based on a number of the author’s peer-reviewed journal and conference papers:

“GraphCT: Multithreaded Algorithms for Massive Graph Analysis”

This article describes the design and performance of an end-to-end graph analytics suite for the Cray XMT. It is featured in Chapter 3.

“A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets”

This paper describes a lock-free parallel algorithm for estimating betweenness centrality on the Cray XMT, and is the basis for Section 3.3.

“Generalizing k-Betweenness Centrality Using Short Paths and a Parallel Multithreaded Implementation”

This paper describes an extension of betweenness centrality to increase robustness in the measure. It is featured in Section 3.3.

“STINGER: High Performance Data Structure for Streaming Graphs”

This paper describes a parallel data structure for streaming, semantic graphs and techniques for obtaining millions of updates per second on commodity and HPC hardware. It is detailed in Chapter 4.

“Massive Streaming Data Analytics: A Case Study with Clustering Coefficients”

This paper describes several algorithmic approaches to tracking clustering coefficients on an edge stream of insertions and deletions. It is the basis for Section 4.3.

“Tracking Structure of Streaming Social Networks”

This paper describes an algorithm and heuristics for efficiently updating the connected components of the graph given a stream of edge insertions and deletions. It is the basis for Section 4.4.

“Investigating Graph Algorithms in the BSP Model on the Cray XMT”

This paper implements several graph algorithms in the BSP model and describes parallel performance effects on the Cray XMT comparing against highly-tuned shared memory implementations. It contributes to Section 5.2.

CHAPTER II

ORIGIN AND HISTORY OF THE PROBLEM

It is said that the beginning of *graph theory* came about as the result of Leonhard Euler's 18th Century study of the seven bridges of Königsberg. Defining each area of land as a vertex and each bridge as an edge, Euler constructed a graph and later proved that all seven bridges could not be traversed without traversing one more than once. Today, modern graph theory permeates many application areas including business, biology, sociology, and computer science.

In graph theory, a graph $G = (V, E)$ is composed of a set of vertices V and a set of edges E . An edge $e \in E$ consists of a pair of vertices $\langle u, v \rangle \mid u, v \in V$ representing a relationship between those two vertices. The edges in a graph can be directed or undirected. In many practical implementations, an undirected graph is a special case of a directed graph in which for each edge $\langle u, v \rangle$, the corresponding edge $\langle v, u \rangle$ is required to exist. A graph in which each edge is assigned a type is sometimes referred to as a *semantic graph*. In this case, the edge is represented by a 3-tuple $\langle u, v, w \rangle$ in which u and v are nouns and w represents the verb relating one to the other. Vertices can also be assigned types. In some applications, the type may be referred to as a weight, a color, or a timestamp. The vertex or edge type may be a vector. A *multigraph* is a graph in which an edge between two vertices can be non-unique. A *hypergraph* is a special type of graph in which a single edge relates more than two vertices.

For several decades, our increasingly digital world has produced overwhelming quantities of raw, unstructured data that can be analyzed as a massive graph. Genome sequences, protein interaction networks, and digital medical records can be used

to better understand human disease. Financial markets handling billions of shares changing hands each day must identify and handle fraudulent activity. Online social networks, with upwards of one billion active users, can yield invaluable data about social, retail, and health trends, among others.

The size and speed at which data is created necessitates the use of parallel computing to conduct analyses. However, parallel computing reveals a challenging new side of graph theory. Let us consider a textbook example. Given an undirected graph G with vertex set V and edge set E and a source vertex $v \in V$. The objective is to find all vertices reachable from v . In classical graph theory, we can do so using a breadth-first search or a depth-first search. The two algorithms produce the same result, albeit in a different order, with the same work complexity. In parallel graph algorithms, the breadth-first search can be parallelized in a level-synchronous fashion: all of the vertices in level d of the graph can search their neighbors and identify those vertices at level $d + 1$ in parallel [1]. Alternatively, one could start several partial breadth-first searches in the graph and combine their results to obtain the final answer [2]. There is no known parallel algorithm for depth-first search.

This simple example illustrates a common theme in parallel graph analysis. The quantity of parallelism available in a graph algorithm can vary widely. Some classical algorithms contain little or no parallelism, and new algorithms must be developed that can scale to large parallel systems and large data sets.

2.1 Relevant Applications

2.1.1 Computational Biology

The size and scope of sequencing the human genome required computational tools to complete. High-speed “shotgun sequencing” technology is currently used for high throughput analysis. This technique produces many small sequences that must be stitched together. A number of tools have been proposed. Velvet is a package of

algorithms for assembling so-called “shotgun sequences” using de Bruijn graphs [3]. A vertex in the graph represents an overlapping k -mer with directed edges to other overlapping k -mers. A read is then mapped to a traversal of the graph.

With the ability to sequence entire genomes, it becomes possible to infer evolutionary events through phylogenetic tree construction. One method constructs a maximum likelihood tree based on DNA sequences and a probabilistic model of evolution [4]. Constructing the tree from multiple gene rearrangement events can be formulated as a series of Traveling Salesperson Problems [5], which are NP-complete. Spectral methods of graph partitioning have also been applied to phylogenetic tree construction [6].

The study of protein-protein interaction networks in yeast reveals a power-law distribution in connectivity. Using vertices to represent a protein and edges to represent a producer-consumer interaction, Barabasi noted a correlation between vertex degree and the change in phenotype when the protein is removed [7]. It was later discovered that a relationship existed between betweenness centrality [8] of a vertex in the network and the essentiality of a protein [9]. Bader and Madduri used a parallel, multithreaded algorithm for betweenness centrality to rank proteins in the human protein interaction network [10], obtaining a 16x speedup with 32 threads.

2.1.2 Business Analytics

Graph theory has the potential to impact large-scale business operations. Guimerà analyzed the worldwide air transportation network, discovering an average shortest path distance of 4.4 [11]. The clustering coefficient for the network was significantly higher than for a random graph, providing further evidence of a small-world property. A betweenness centrality analysis found that Anchorage, Alaska was in the top 25 most central cities, but had few connections to other cities. The networks used in this study consisted of 3,883 vertices and 27,051 edges.

Other studies have looked at road networks [12], railway networks [13], and cargo ship networks [14]. Among shipping routes, betweenness centrality indicates that some ports are highly central for all types of vessels (i.e. the Suez Canal), while others are only central for a specific type of ship. The authors performed a triad motif study that found transitive triads to be prominent with few intransitive motifs. This suggests that the routes are chosen to have robustness and resilience to weather and equipment failure.

The study of transportation networks has societal impact above and beyond the business case. Invasive organisms can be transported by boat through ballast tanks or inside shipping containers from one continent to another [14]. Colizza et. al demonstrated that the airline transportation network, when combined with population census data, could be used to understand the pattern of worldwide disease spread [15].

Beyond transportation networks, graph theory can provide deeper insight into infrastructure reliability. When applied to the North American electric power grid, new forms of betweenness centrality enable power distribution companies to understand the effect of multiple failures [16]. In this particular work, the Cray XMT is a good fit for the problem and achieves strong scalability up to 64 processors.

2.1.3 Security

The growing corpus of data that is available to law enforcement makes finding the proverbial “needle in a haystack” ever more difficult. Turning that data into actionable information quickly is also a challenge. Approximate subgraph isomorphism and more complex social network analysis metrics have proven useful for detecting anomalous activity in terrorist networks [17]. Yet still, better algorithms and scalable systems are needed to reduce the number of false positives considered. In many cases, optimizing the criteria of interest is NP-complete, motivating the development of fast

heuristics.

The link graph of the world wide web is known to have a power-law distribution in the number of neighbors and a small diameter, although estimates of the diameter vary widely [18, 19]. This structure is thought to provide robustness to the removal of pages [20]. However, it has been shown that the network is still vulnerable to an intentional attack by an adversary, despite its scale-free structure [21].

Graph analytics in this application area look for anomalous events and structures. In a large network with many thousands of systems and applications, an anomalous event may be implicated in a compromise or pending attack. New algorithms can help improve spam and fraud detection for corporate systems using both the content of the message and the pattern of activity, which is language-agnostic.

2.1.4 Social Networks

The growth of massive social networks on the Internet in recent years is staggering. Facebook has more than 1 billion users, over half of which are active daily [22]. Twitter has tens of millions of users, and the blogosphere has an estimated hundreds of millions of English language blogs. In each case, the network contains both topological information (actors and links) as well as a rich semantic network of interactions. If the topology information of Facebook alone was represented in compressed sparse row (CSR) format with edge weights using 64-bit data types, the data structure alone would cost over 1.9 TiB of memory. The scale of these social networks necessitates specialized computer architecture and massively parallel algorithms for analysis.

In a recent paper (see [23]), we formed a social graph from posts on Twitter relating to a national influenza outbreak and a local historic flood event. Applying social network analytics to these datasets, we determined that during a crisis period, Twitter behaves primarily as a broadcast medium for government and the news media to spread information to the public. However, our analysis showed that some local

individual users were equally important in disseminating content. A very small subset of the posts were conversational in nature, but remained on topic.

A key challenge is mapping individuals to distinct or overlapping communities based on the structure of the network. The classic example in the literature is Zachary’s karate club, in which the interactions of 34 people are observed and recorded over two years [24]. Graph partitioning algorithms generally agree that two communities exist. Girvan and Newman proposed using betweenness centrality to remove links from the network until communities formed [25], although this approach is computationally-intensive. More recently, Clauset, Newman, and Moore developed an agglomerative clustering approach that maximizes modularity in the clusters [26]. We provided a generalized parallel framework for agglomerative clustering, based on CNM, that is agnostic to the scoring method [27].

Until recently, social networks, such as the karate club, were analyzed as a complete picture. If the structure of the graph changed significantly, the analytics would be recomputed. Current social networks, such as Facebook and Twitter, are too large and the ingest rate of new edge data is too high to sustain such a static “snapshot” analysis. In many cases, little or no change in the metric of interest is generated by a new edge insertion or deletion. In order to exploit this locality of sensitivity, we developed methodologies for tracking the clustering coefficients [28] and connected components [29] of a streaming graph. We are also developing a parallel data structure called STINGER [30] that is the foundation for our temporal analytics. We will briefly describe these contributions in Chapter 4.

2.2 Architectures and Programming Models

Real world networks challenge modern computing in several ways. These graphs typically exhibit “small-world” [31] properties such as small diameter and skewed degree distribution. The low diameter implies that all reachable vertices can be

found in a small number of hops. A highly skewed degree distribution, where most vertices have a few neighbors and several vertices have many neighbors, will often lead to workload imbalance among threads. One proposed solution is to handle high- and low-degree vertices separately; parallelize work across low-degree vertices and within high-degree vertices. A runtime system must be able to handle dynamic, fine-grained parallelism among hundreds of thousands of threads with low overhead. Executing a breadth-first search from a particular vertex quickly consumes the entire graph. The memory access pattern of such an operation is unpredictable with little spatial or temporal locality. Caches are ineffective for lowering memory access latency in this case.

In this section, two very different hardware platforms and programming models for massively parallel applications are discussed. The first, the Cray XMT, is a shared memory supercomputer purpose-built for large graph analytics with terabytes of main memory. The second, the MapReduce cluster, is made up of commodity servers and networks and is able to scale out to petabyte-sized datasets.

2.2.1 Cray XMT

The Cray XMT [32] is a supercomputing platform designed to accelerate massive graph analysis codes. The architecture tolerates high memory latencies using massive hardware multithreading. Fine-grained synchronization constructs are supported through full-empty bits as well as atomic fetch-and-add instructions. A large fully shared memory enables the analysis of graphs on the order of one billion vertices using a well-understood programming model.

Each Threadstorm processor within a Cray XMT contains 128 *hardware streams*. Streams may block temporarily while waiting for a long-latency instruction, such as a memory request, to return. The processor will execute one instruction per cycle from hardware streams that have instructions ready to execute. The Cray XMT

does not require locality to obtain good performance. Instead, latency to memory is tolerated entirely by hardware multithreading, making this machine a good candidate for memory-intensive codes like those found in graph analysis.

The Cray XMT located at Pacific Northwest National Lab contains 128 Threadstorm processors running at 500 MHz. These 128 processors support over 12 thousand hardware thread contexts. The globally addressable shared memory totals 1 TiB. Memory addresses are hashed globally to break up locality and reduce hot-spotting. The Cray XMT2 is a next-generation system located at the Swiss National Supercomputing Centre and contains 64 Threadstorm processors with 2 TiB globally shared memory.

A global shared memory provides an easy programming model for the graph application developer. Due to the “small-world” nature of the graphs of interest, finding a balanced partition of vertices or edges across multiple distinct memories can be difficult. The Cray XMT has proved useful in a number of irregular applications including string matching [33], document clustering [34], triangle counting [35], hash tables [36], static graph analysis [16, 23, 27, 37, 38] and streaming graphs [28, 29].

2.2.2 MapReduce

MapReduce is a parallel programming model for large data computations on distributed memory clusters [39]. MapReduce jobs consist of two functions: a mapper and a reducer. The mapper takes (key, value) pairs as input and produces an intermediate set of (key, value) pairs. The intermediate (key, value) pairs are shuffled among the reducers according to key. The reducers combine the values for a given key into a single output.

While developed at Google initially for faster search indexing using PageRank, MapReduce has attracted the attention of the large graph analytics community for its ability to perform operations on petabyte-scale datasets. HADI [40] is a MapReduce

algorithm for computing the effective diameter of a graph, or the minimum number of hops such that 90 percent of connected pairs of vertices can reach each other. The algorithm iteratively expands a neighborhood around each vertex. The authors report a running time of 3.5 hours to calculate the effective diameter of a Kronecker graph with two billion edges and 177 thousand vertices.

Cohen gives algorithms in the MapReduce model for triangle counting and connected components in [41]. The connected components algorithm begins by mapping each vertex to its own component, then iteratively merging components according to the edges in the graph, similar to the Shiloach-Vishkin classical algorithm [42]. In his conclusion, Cohen points out that the MapReduce model leaves open the possibility of moving the entire graph from one processing node to another. Most graphs consist of one giant connected component and many small components on the fringe. In this case, indeed much of the graph edges will be processed and subsequently dropped (by the relabeling) by one machine.

Chierichetti et. al parallelize a classical greedy algorithm for the NP-hard max-cover problem using MapReduce with similar performance and approximation guarantees [43]. A MapReduce algorithm for clustering coefficients is given in [44]. On a MapReduce cluster with 32 nodes and gigabit Ethernet interconnect, the authors calculate the clustering coefficient of a web graph with 1.63 million vertices and 5.68 million edges in 160 seconds. The literature contains many other examples of simple graph analytics computed on distributed memory clusters with MapReduce whose performance leaves room for improvement.

The core strength of MapReduce is its ability analyze truly massive datasets using commodity hardware. However, the programming model gives the programmer no control over data movement or locality. As a result, network bandwidth is often the bottleneck in MapReduce jobs. Further, current implementations, such as Yahoo's Hadoop [45] require intermediate data to be written to disk between phases and

iterations. Some graph algorithms have the ability to generate a large amount of intermediate data between iterations (such as breadth-first search).

One proposed solution is to develop a hybrid system that leverages the MapReduce model to store the raw data and build the graph, but does the graph analytics in a massively parallel multithreaded environment [46]. The authors considered a scale-free input graph with 4.3 billion vertices and 275 billion edges. The experiment involved extracting a subgraph covering 10 percent of the vertices and finding shortest paths between 30 pairs of vertices in the subgraph. The Hadoop cluster took 24 hours to do the extraction and 103 hours to perform the shortest paths computation. The multithreaded system, a Sun UltraSPARC T2, performed the shortest paths computation on the subgraph in several seconds, taking only about 50 minutes of time to transfer the graph from one system to the other.

2.3 High Performance Graph Research

Much of the prior work on graph analysis software was done in the context of sequential algorithms on desktop workstations. Many such packages exist, such as Pajek [47], UCINET [48], igraph [49], Tulip [50] and Cytoscape [51], among others. These packages contain a number of different analytics that can be computed, and many are also able to visualize the input graph. Efficient visualization of scale-free networks is a topic that is still open for research. Due to resource requirements of some algorithms, these packages are limited in the size of the input graph they can process; usually ten thousand to several million edges.

In order to analyze current graph data with billions of edges, there are several ongoing research efforts in high performance graph analysis. Each of the following projects targets a different programming model and hardware architecture.

2.3.1 Parallel Boost Graph Library

The Parallel Boost Graph Library (PBGL) [52] is a C++ library for distributed memory graph computations. The API is similar to, but not fully compatible with, that of the Boost Graph Library. PBGL is designed for distributed memory clusters that use the Message Passing Interface (MPI). The edges of the graph are stored in an adjacency list that is distributed among the cluster nodes. PBGL supports edge and vertex weights, or properties, that are distributed with the graph. Algorithms such as breadth-first search, Dijkstra single source shortest path, and connected components are provided. The authors report scalable performance up to about 100 processors. Graph processing using distributed memory clusters can be challenging as edges and vertices may not be easily partitioned among the cluster nodes and the cost and volume of communication and synchronization may be quite high. Recent research by the PBGL team suggests that the use of *active messages* in graph algorithms may help to reduce the communication cost in distributed memory systems [53].

2.3.2 GraphCT

GraphCT [54] is a framework for developing parallel and scalable static graph algorithms on multithreaded platforms. The foundation of GraphCT is a modular kernel-based design using efficient data representations in which an analysis workflow can be expressed through a series of function calls. All functions are required to use a single graph data representation. While other frameworks use different data structures for different input graphs or analysis kernels, the use of a single common data structure enables plug-and-play capability as well as ease of implementation and sharing new kernels. Basic data input/output as well as fundamental graph operations such as subgraph extraction are provided to enable domain scientists to focus on implementing high-level analyses. A wide variety of multithreaded graph algorithms are provided including clustering coefficients, connected components, betweenness

centrality, k -core, and others, from which workflows can easily be developed. Analysis can be conducted on unweighted and weighted graphs, undirected and directed. GraphCT can be built on the Cray XMT with XMT-C or on a commodity workstation using OpenMP.

2.3.3 Sandia Multithreaded Graph Library

Sandia’s Multithreaded Graph Library (MTGL) [55] is a C++ library for implementing graph routines on multithreaded architectures, particularly the Cray XMT. MTGL uses the notion of a *visitor class* to describe operations that take place when a vertex is visited, such as during a breadth-first search.

MTGL is suited for directed and undirected graphs, graphs with types, and multigraphs. On the Cray XMT, important primitives such as full-empty bit semantics and atomic fetch-and-add are wrapped with new MTGL designations. MTGL successfully implements scalable versions of popular kernels such as breadth-first search, connected components, and *st*-connectivity.

For portability to commodity shared-memory systems, such as x86 workstations, MTGL will run atop Sandia’s Qthreads user-level threading library [56]. Qthreads emulates the full-empty bit memory semantics on commodity systems and takes advantage of hardware support for atomic fetch-and-add when available. On multicore systems, Qthreads provides lightweight threads that are required for the fine-grained parallelism of graph applications.

2.3.4 Knowledge Discovery Toolbox

The Knowledge Discovery Toolbox (KDT) [57] enables high performance graph analysis in Python by leveraging highly-tuned sparse linear algebra kernels. The linear algebra kernels are written as a C-language library, enabling support for MPI and accelerators. In distributed mode, the adjacency matrix is striped across nodes using

a 1D vertex partitioning. Linear algebra primitives are exposed to the Python environment as functions and operators. In this manner, graph algorithms that are easily expressed as matrix computations are programmed in a straightforward manner and scaled to large clusters with low complexity.

One drawback to this approach is the overhead of interactions between Python and the C-language library modules. The frequency of data and control transfer between Python and C correlates with performance. For example, filtering edges with a user-specified filter in Python exhibits lower performance than a built-in filter in C because an interaction is required for each edge. While this approach is very powerful on a number of graph algorithms, it is not yet well-understood how to apply this approach to all graph algorithms, including those for streaming updates.

2.3.5 Google Pregel

Pregel [58] is a distributed graph processing system with a C++ API developed by Google. To avoid issues of deadlock and data races, Pregel uses a bulk synchronous parallel (BSP) model of computation. A graph computation is broken up into a sequence of iterations. In each iteration, a vertex is able to 1) receive messages from the previous iteration, 2) do local computation or modify the graph, and 3) send messages to vertices that will be received in the next iteration. Similar to MapReduce in many ways, chains of iterations are used to solve a graph query in a fault-tolerant manner across hundreds or thousands of distributed workstations. Unlike MapReduce, however, vertices in Pregel can maintain state between iterations, reducing the communication cost.

To support this model of computation in a distributed cluster environment, Pregel assigns vertices to machines along with all of their incident edges. A common operation is for a vertex to send a message to its neighbors, which are readily available, but the model allows for sending messages to any vertex that is known by the sender

through other means. By default, the assignment of vertex to machine is based on a random hash function yielding a uniform distribution of the vertices. Real-world graphs, however, have the scale-free property. In this case, the distribution of edges will be uneven with one or several machines acquiring high degree vertices, and therefore a disproportionate share of the messaging activity.

CHAPTER III

GRAPHCT

The vast quantity of data being created by social networks [22], sensor networks [59], healthcare records [60], bioinformatics [7], computer network security [21], computational sciences [61], and many other fields offers new challenges for analysis. When represented as a graph, this data can fuel knowledge discovery by revealing significant interactions and community structures. Current network analysis software packages (e.g. Pajek [47], R (igraph) [49], Tulip [50], UCInet [48]) can handle graphs up to several thousand vertices and a million edges. These applications are limited by the scalability of the supported algorithms and the resources of the workstation. In order to analyze today's graphs and the semantic data of the future, scalable algorithms and machine architectures are needed for data-intensive computing. GraphCT [62] is a collection of new parallel and scalable algorithms for static graph analysis. These algorithms, running atop multithreaded architectures such as the Cray XMT, can analyze graphs with hundreds of millions of vertices and billions of edges in minutes, instead of days or weeks. GraphCT is able to, for the first time, enable analysts and domain experts to conduct in-depth analytical workflows of their data at massive scales.

The foundation of GraphCT is a modular, kernel-based design using efficient data representations in which an analysis workflow can be expressed through a series of function calls. The high-level framework is illustrated in Figure 1. All functions are required to use a single graph data representation. The use of a single common data structure enables plug-and-play capability as well as ease of implementation and

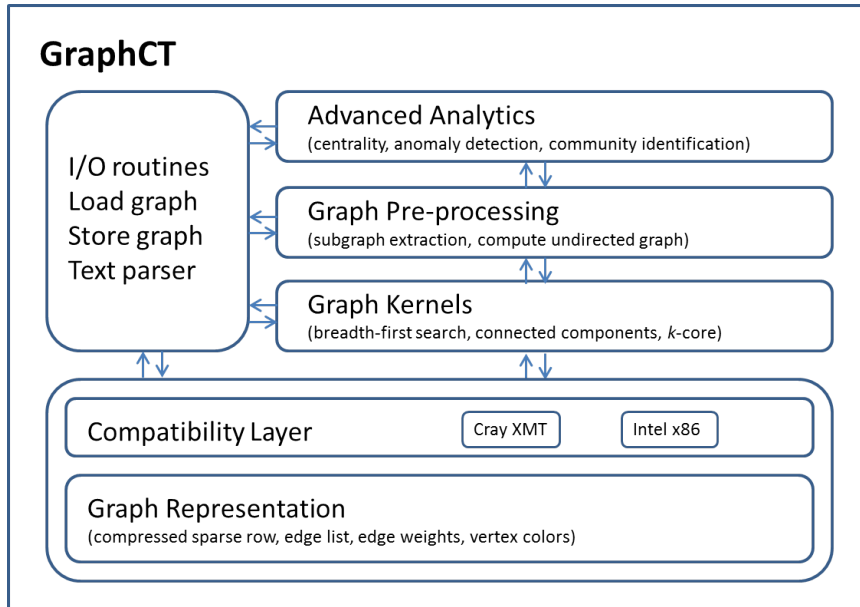


Figure 1: GraphCT is an open source framework for developing scalable multi-threaded graph analytics in a cross-platform environment.

sharing new kernels. Basic data input/output as well as fundamental graph operations such as subgraph extraction are provided to enable domain scientists to focus on conducting high-level analyses. A wide variety of multithreaded graph algorithms are provided including clustering coefficients, connected components, betweenness centrality, k -core, and others, from which workflows can be easily developed. Figure 2 illustrates an example workflow. Analysis can be conducted on unweighted and weighted graphs, undirected and directed. Limited sections of GraphCT are parallelized for parallel platforms other than the Cray XMT.

A number of software applications have been developed for analyzing and visualizing graph datasets. Among them, Pajek is one of the most widely used, along with R (igraph), Tulip, UCInet, and many others [47, 49, 50, 48]. While each application has its differences, all are limited by the size of workstation main memory and do not take advantage of parallel systems. Pajek has been known to run complex graph analytics on inputs with up to two million vertices, but many other applications are limited to tens or hundreds of thousands of vertices.

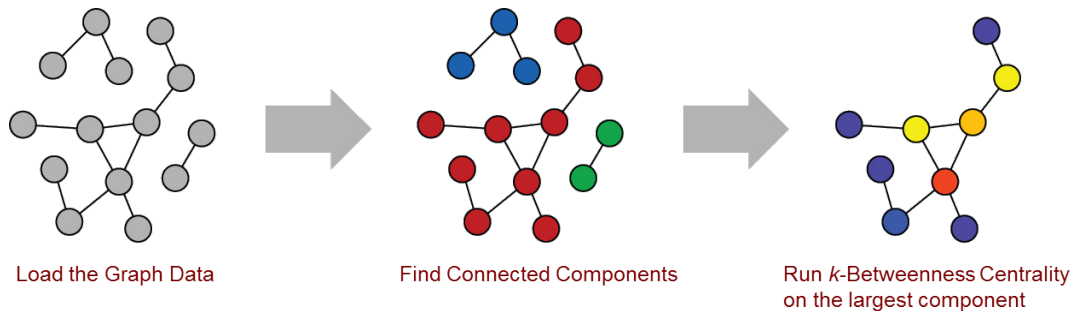


Figure 2: An example user analysis workflow in which the graph is constructed, the vertices are labeled according to their connected components, and a single component is extracted for further analysis using several complex metrics, such as betweenness centrality.

Table 1: Graph analysis packages and frameworks currently under development.

Package	Interface	Parallel	Memory	$O(Edges)$	Analytics	Frameworks
Pregel [58]	C++	X	Distributed on-disk	127 billion		X
MTGL [55]	C++	X	Shared (Cray XMT)	35 billion		X
GraphCT [54]	C	X	Shared (Cray XMT)	17 billion	X	X
PBGL [52]	C++	X	Distributed in-memory	17 billion		X
KDT [57]	Python	X	Distributed in-memory	8 billion	X	X
Pegasus [63]	Hadoop	X	Distributed on-disk	6.6 billion		X
NetworkX [64]	Python		Shared	100 million	X	
SNAP [65]	C	X	Shared	32 million	X	
Pajek [47]	Windows		Shared	16 million	X	
igraph [49]	R		Shared	Millions	X	

Table 1 describes several graph analytic applications and several high performance graph frameworks that are under active development. For each project, we list the largest graph size computation published in the literature by the project developers. Of the packages that include an end-to-end analytics solution, GraphCT is able to process the largest graphs.

The development of new scalable algorithms and frameworks for massive graph analysis is the subject of many research efforts. The Parallel Boost Graph Library (PBGL) [52] is a C++ library for distributed memory graph computations. The API is similar to that of the Boost Graph Library. The authors report scalable performance up to about 100 processors. Distributed memory graph processing often requires partitioning and data replication, which can be challenging for some

classes of graphs. Google’s Pregel [58] uses a MapReduce-like programming model for describing vertex-centric graph computations on large, distributed memory clusters. Buluç and Madduri have demonstrated high performance techniques for scaling breadth-first search on distributed memory supercomputers [66].

SNAP [65] is an open source parallel library for network analysis and partitioning using multicore workstations. It is parallelized using OpenMP and provides a simple API with support for very large scale graphs. It is one of the only libraries that provides a suite of algorithms for community detection. The Knowledge Discovery Toolbox (KDT) [57] enables high performance graph analysis in Python by leveraging highly-tuned sparse linear algebra kernels.

Sandia’s Multithreaded Graph Library (MTGL) [55] is a C++ library for implementing graph applications on multithreaded architectures, particularly the Cray XMT. MTGL uses the notion of a “visitor” class to describe operations that take place when a vertex is visited, such as during a breadth-first search.

Other approaches to large graph problems include the NoSQL graph databases used by the semantic web community. These graph databases implement RDF triplestores that support ACID properties. They lack relational database schemas, but include query languages such as SPARQL [67]. Also, WebGraph is a graph compression technique for large graphs generated through web crawls [68].

Given the immense size in memory of the graphs of interest, it is not possible to store a separate representation for each analysis kernel. Since the key capability of GraphCT is running a number of analytics against an unknown data source, we employ a simple, yet powerful framework for our computation. Each kernel implementation is required to use a common graph data structure. By using the same data structure for each kernel, all kernels can be run in succession (or even in parallel if resources allow) without the need to translate the graph between data structures. In the client/server model, a GraphCT server process loads the graph into memory and

shares it in read-only mode with all client analytic processes, amortizing the time required to load the data and generate the graph. The results of one computation can easily influence the next computation, such as the extraction of one or more connected components for more in-depth study.

Efficient representation of networks is a well-studied problem with numerous options to choose from depending on the size, topology, degree distribution and other characteristics of a particular graph. If these characteristics are known *a priori*, one may be able to leverage this knowledge to store the graph in a manner that will provide the best performance for a given algorithm. Because we are unable to make any assumptions about the graph under study and will be running a variety of algorithms on the data, we must choose a representation that will provide adequate performance for all types of graphs and analytics.

To facilitate scaling to the sizes of massive datasets previously described, GraphCT utilizes the massive shared memory and multithreading capabilities of the Cray XMT. Large planar graphs, such as road networks, can be partitioned with small separators and analyzed in distributed memory with good computation-to-communication ratios at the boundaries. Graphs arising from massive social networks, on the other hand, are challenging to partition and lack small separators [65, 69]. For these problems, utilizing a large, global shared memory eliminates the requirement that data must be evenly partitioned. The entire graph can be stored in main memory and accessed by all threads. With this architectural feature, parallelism can be expressed at the level of individual vertices and edges. Enabling parallelism at this level requires fine-grained synchronization constructs such as atomic fetch-and-add and compare-and-swap.

The Cray XMT offers a global shared memory using physically distributed memories interconnected by a high speed, low latency, proprietary network. Memory addresses are hashed to intentionally break up locality, effectively spreading data throughout the machine. As a result, nearly every memory reference is a read or

```

#pragma mta assert no dependence
for (k = 0; k < numEdges; k++) {
    int i = sV[k];
    int j = eV[k];
    if (D[i] < D[j])
        D[j] = D[i];
}

```

Figure 3: This MTA pragma instructs the Cray XMT compiler that the loop iterations are independent.

write to a remote memory. Graph analysis codes are generally a series of memory references with very little computation in between, resulting in an application that runs at the speed of memory and the network.

On the Cray XMT, hardware multithreading is used to overcome the latency of repeated memory accesses. A single processor has 128 hardware contexts and can switch threads in a single cycle. A thread executes until it reaches a long-latency instruction, such as a memory reference. Instead of blocking, the processor will switch to another thread with an instruction ready to execute on the next cycle. Given sufficient parallelism and hardware contexts, the processor’s execution units can stay busy and hide some or all of the memory latency.

Since a 128-processor Cray XMT contains about 12,000 user hardware contexts, it is the responsibility of the programmer to reveal a large degree of parallelism in the code. Coarse- as well as fine-grained parallelism can be exploited using Cray’s parallelizing compiler. The programmer inserts `#pragma` statements to assert that a loop’s iterations are independent (see Figure 3). Often iterations of a loop will synchronize on shared data. To exploit this parallelism, the Cray XMT provides low-cost, fine-grained synchronization primitives such as full-empty bit synchronization and atomic fetch-and-add [70]. Using these constructs, it is possible to expose fine-grained parallelism, such as operations over all vertices and all neighbors, as well as coarse-grained parallelism, such as multiple breadth-first searches in parallel.

The design model for GraphCT dictates that all analysis kernels should be able to read from a common data representation of the input graph. A function can allocate its own auxiliary data structures (queues, lists, etc.) in order to perform a calculation, but the edge and vertex data should not be duplicated. This design principle allows for efficient use of the machine’s memory to support massive graphs and complex queries. We refer the reader to [71] for an in-depth study of graph algorithms.

The data representation used internally for the graph is an extension based on compressed sparse row (CSR) format. In CSR, contiguous edges originating from the same source vertex are stored by destination vertex only. An offset array indicates at which location a particular vertex’s edges begin. The common access pattern is a two-deep loop nest in which the outer loop iterates over all vertices, and the inner loop identifies the subset of edges originating from a vertex and performs a computation over its neighbors. We build upon the CSR format by additionally storing the source vertex, thus also expressing an edge list directly. Although redundant, some kernels can be expressed efficiently by parallelizing over the entire edge list, eliminating some load balance issues using a single loop. In this way, the internal graph data representation allows for the easy implementation of edge-centric kernels as well as vertex-centric kernels.

For weighted graphs, we store the weight of each edge represented with a 64-bit integer. We allocate an additional array with length of the number of vertices that each function can use according to its own requirements. In some cases, such as breadth-first search, a kernel marks vertices as it visits them. This array can be used to provide a coloring or mapping as input or output of a function. This coloring could be used to extract individual components, as an example.

In this format, we can represent both directed and undirected graphs. The common data representation between kernels relieves some of the burden of allocating frequently used in-memory data structures. With the graph remaining in-memory

between kernel calls, we provide a straightforward API through which analytics can communicate their results.

GraphCT supports multiple client processes connecting to multiple server processes that store graph data. Each server process computes a graph data structure in shared memory. The server process advertises its graph with a unique identifier. Clients reference the unique identifier in lieu of a graph file on disk. A client process maps the shared graph into its own memory space and computes on it normally. This approach can amortize the cost of building a large graph, allowing many analytics to be run in parallel.

The connected components of the graph is the maximal set of vertices such that any vertex is reachable from any vertex in the component. If two vertices are in the same component, then there exists a path between them. Likewise, if two vertices reside in different components, a search from one vertex will not find the other. If the connected components of the graph are known, determining the st connectivity for a pair of vertices can be calculated easily.

In Section 3.1 we will offer in-depth coverage of the algorithm, implementation, and performance of our connected components routine. We use a shared memory version of the classical Shiloach and Vishkin algorithm [42]. On the Cray XMT, we determine the connected components of a scale-free, undirected graph with 135 million vertices and 2 billion edges in about 15 seconds.

Clustering coefficients measure the density of closed triangles in a network and are one method for determining if a graph is a small-world graph [31]. For undirected graphs, we can compute the global clustering coefficient, which is a single number describing the entire graph, or the local clustering coefficients, which is a per-vertex measure of triangles. For directed graphs, several variations have been proposed and we have adopted the transitivity coefficients, which is a natural extension of the local clustering coefficient.

Section 3.2 contains a detailed case study of our implementation on the Cray XMT and performance results on large synthetic networks.

When the nature of the input graph is unknown, the degree distribution is often a metric of interest. The degree distribution will indicate how sparse or dense the graph is, and the maximum degree and variance will indicate how skewed the distribution is. A skewed distribution may actually be a power-law distribution or indicate that the graph comes from a data source with small-world properties. From a programmer’s perspective, a large variance in degree relative to the average may indicate challenges in parallelism and load balance.

The maximum degree, average degree, and variance are calculated using a single parallel loop and several accumulators over the vertex offset array. On the Cray XMT, the compiler is able to automatically parallelize this loop. Given a frequency count, GraphCT produces a histogram of values and the distribution statistics.

The diameter of the graph is an important metric for understanding the nature of the input graph at first glance. If interested in the spread of disease in an interaction network, the diameter is helpful to estimate the rate of transmission and the time to full coverage. Calculating the diameter exactly requires an all-pairs shortest path computation, which is prohibitive for the large graphs of interest.

In GraphCT, we estimate the diameter by random sampling. Given a fixed number of source vertices (expressed as a percentage of the total number of vertices), a breadth-first search is executed from each chosen source. The length of the longest path found during that search is compared to the global maximum seen so far and updated if it is longer. With each sample we more closely approximate the true diameter of the graph. Ignoring the existence of long chains of vertices, we can obtain a reasonable estimate with only a small fraction of the total number of breadth-first searches required to get an exact diameter [72]. However, GraphCT leaves the option of the number of samples to the user, so an exact computation can be requested.

Obtaining a reasonable estimate of the graph diameter can have practical consequences for the analysis kernels. A kernel running a level-synchronous breadth-first search will require a queue for each level. The total number of items in all of the queues is bounded by the number of vertices, but the number of queues is bounded by the graph diameter. If the diameter is assumed to be on the order of the square root of the number of vertices (a computer network perhaps) and the same kernel is run on an input graph where the diameter is much larger (a road network), the analysis will run out of memory and abort. On the other hand, allocating a queue for the worst-case case scenario of a long chain of vertices is overly pessimistic. By quickly estimating the diameter of the graph using a small number of breadth-first searches, we can attempt to allocate the “right” amount of memory upfront.

Given that the graphs of interest are so large as to require machines with terabytes of main memory, we expect the input data files to also be of massive size. GraphCT is the only graph application for the Cray XMT that parses input text files in parallel in the massive shared memory.

GraphCT supports the common DIMACS format in which each line consists of a letter “a” (indicating it is an edge), the source vertex number, destination vertex number, and an edge weight. To leverage the large shared memory of the Cray XMT, we copy the entire file from disk into main memory. In parallel, each thread obtains a block of lines to process. A thread reserves a corresponding number of entries in an edge list. Reading the line, the thread obtains each field and writes it into the edge list. Once all threads have completed parsing the text file, it is discarded and the threads cooperatively build the graph data structure. In this manner, we are able to process text files with sizes ranging in the hundreds of gigabytes in just a few seconds.

For instances when real data is not available with the scale or characteristics of interest, GraphCT is able to provide graph generators that will provide an output file on disk that can be read in for kernel testing. GraphCT includes an implementation

of the RMat graph generator [73], which was used in the DARPA High Productivity Computing Systems (HPCS) Scalable Synthetic Compact Applications benchmark #2 (SSCA2). This generator uses repeated sampling from a Kronecker product to produce a random graph with a degree distribution similar to those arising from social networks. The generator takes as input the probabilities a , b , c , and d which determine the degree distribution, the number of vertices (must be a power of two), and the number of edges. Duplicate and self edges are removed.

A k -core is the subgraph induced when all vertices with degree less than k are removed. The k -core can be used to study clustering in networks and as pre-processing for other analytics. We provide functionality that repeatedly removes vertices in a graph with degree less than k until no such vertices remain. The remaining vertices and incident edges are extracted and returned to the user as a new graph data structure that can be passed to further analyses.

In graph partitioning and community detection, a variety of scoring functions have been proposed for evaluating the quality of a cut or community. Among the more popular metrics is modularity and conductance. Modularity is a measure of interconnectedness of vertices in a group. A community with a high modularity score is one in which the vertices are more tightly connected within the community than with the rest of the network. Formally, modularity is defined as:

$$Q = \frac{1}{4m} \sum_{ij} \left(A_{ij} - \frac{k_i k_j}{2m} \right) s_i s_j \quad (1)$$

where i and j are vertices in the graph, m is the total number of edges, k_i is the degree of vertex i , and s_i expresses the community to which vertex i belongs [74].

Given a community mapping of vertices, modularity is calculated using two parallel loops over all vertices. The first calculates the total number of edges in each community. The second loop gives credit for neighbors of vertices that are in the same community and subtracts credit for the external connections. The modularity

score is reported and returned at the end. This function is used as a scoring component of a clustering method. For example, in greedy agglomerative clustering, after each component merge the modularity is evaluated and stored in the merge tree.

Conductance is a scoring function for a cut establishing two partitions. The conductance over a cut measures the number of edges within the partition versus the number of edges that span the partition. Conductance can be applied to both directed and undirected graphs, although the undirected version is simplified. Formally, conductance is defined as:

$$\Phi = \frac{e(S, \bar{S})}{d \min \{|S|, |\bar{S}|\}} \quad (2)$$

where \bar{S} is the set of vertices not in S and $e(S, \bar{S})$ is the number of edges between S and \bar{S} . The total number of edges that could span the cut is expressed as $d|S| = \sum_{v \in S} d(v)$ where $d(v)$ is the degree of vertex v [75]. The value of Φ ranges from 0 to 1. While the formula above is for an unweighted graph, it can be generalized to weighted networks by summing edge weights instead of degrees.

Given an edge cut expressed as a 2-coloring of vertices, the conductance is computed by iterating over all edges. Each edge is placed in one of three buckets: 1) both endpoints belong to the same partition, 2) the endpoints are in partition A and B respectively, or 3) the endpoints are in partition B and A respectively. The total number of items in each bucket is counted and the conductance is computed according to the formula based on the larger of the two partitions.

While the intention of GraphCT is to be offered to developers as a library that they can extend and develop their own analysis workflows, we also offer a simple command line interface for core functionality. The single shot query interface enables a user to load in graph data from disk, perform a single query (such as connected components), and write the results out to disk. While this method does not amortize the transfer cost of the graph, it requires no knowledge of the development environment.

Given these two interfaces, the developer library and the single shot query, it is reasonable to develop something in the middle that will enable a domain scientist or analyst to run a series of simple queries after loading the graph data once. We provide a proof-of-concept scripting interface that can load data from disk, run analytic kernels, and extract subgraphs produced by the analysis. We imagine that in a production environment this script parser would be replaced by a GUI control interface.

Betweenness centrality has proved a useful analytic for ranking important vertices and edges in large graphs. Betweenness centrality is a measure of the number of shortest paths in a graph passing through a given vertex [8]. For a graph $G(V, E)$, let σ_{st} denote the number of shortest paths between vertices s and t , and $\sigma_{st}(v)$ the count of shortest paths that pass through a specified vertex v . The betweenness centrality of v is defined as:

$$BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \quad (3)$$

GraphCT on the 128-processor Cray XMT recorded 606 million traversed edges per second on a scale-free graph with 135 million vertices and 2.14 billion edges. The Knowledge Discovery Toolbox (KDT) implements betweenness centrality on a distributed-memory cluster using Combinatorial BLAS. the authors demonstrate performance results on a scale-free graph with approximately 262,000 vertices and 4 million edges. They report 125 million traversed edges per second using 256 cores (24 cores per node) [57].

Algorithm 1 Parallel multithreaded version of Shiloach-Vishkin algorithm for finding the connected components of a graph.

Input: $G(V, E)$

Output: $M[1..n]$, where $M[v]$ is the component to which vertex v belongs

```
1:  $changed \leftarrow 1$ 
2: for all  $v \in V$  in parallel do
3:    $M[v] \leftarrow v$ 
4: while  $changed \neq 0$  do
5:    $changed := 0$ 
6:   for all  $\langle i, j \rangle \in E$  in parallel do
7:     if  $M[i] < M[j]$  then
8:        $M[M[j]] := M[i]$ 
9:        $changed := 1$ 
10:  for all  $v \in V$  in parallel do
11:    while  $M[v] \neq M[M[v]]$  do
12:       $M[v] := M[M[v]]$ 
```

3.1 Connected Components

Finding the connected components of the graph determines a per-vertex mapping such that all vertices in a component are reachable from each other and not reachable from those vertices in other components. A sampling algorithm may sample vertices according to the distribution of component sizes such that all components are appropriately represented in the sampling. An analysis may focus on just the small components or only the biggest component in order to isolate those vertices of greatest interest.

The Shiloach and Vishkin algorithm [42] is a classical algorithm for finding the connected components of an undirected graph. This algorithm is well suited for shared memory and exhibits per-edge parallelism that can be exploited.

In Algorithm 1, each vertex is initialized to its own unique color (line 3). At each step, neighboring vertices greedily color each other such that the vertex with the lowest ID wins (lines 7 and 8). The process ends when each vertex is the same color as its neighbors. The number of steps is proportional to the diameter of the graph,

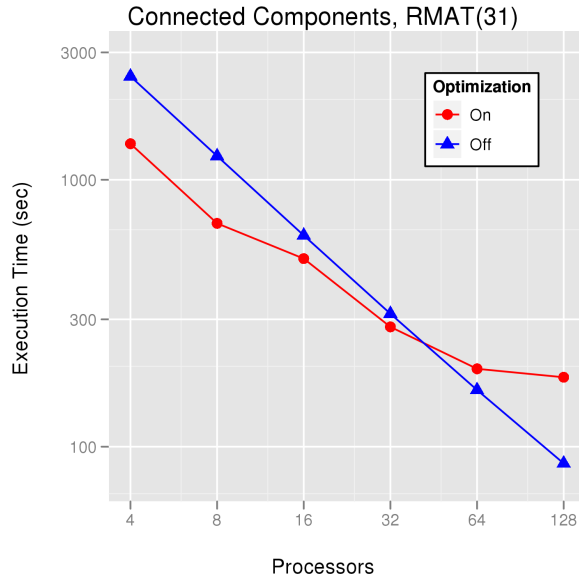


Figure 4: Scalability of Shiloach-Vishkin connected components with and without tree-climbing optimization on the Cray XMT. The input graph is scale-free with approximately 2 billion vertices and 17 billion edges. The speedup is 113x on 128 processors.

so the algorithm converges quickly for small-world networks.

Using the fine-grained synchronization of the Cray XMT, the colors of neighboring vertices are checked and updated in parallel. A shared counter recording the number of changes is updated so as to detect convergence.

In lines 10 through 12 of Algorithm 1, each vertex climbs the component tree, relabeling itself. This optimization can reduce the number of iterations required. In Figure 4, we plot the execution time for an RMAT graph with 2 billion vertices and 17 billion edges as a function of the number of Cray XMT processors. At low processor counts, the optimization on lines 10 through 12 shortens the execution time. However, it creates a memory hotspot and the additional contention at large processor counts produces a less scalable implementation. Removing lines 10 through 12 from the algorithm results in a 113x speedup on 128 processors.

In Table 2, execution times for connected components on several massive undirected graphs using a 128-processor Cray XMT are presented. The first graph is a

Table 2: Running times in seconds for connected components on a 128-processor Cray XMT.

Name	$ V $	$ E $	P=16	P=32	P=64	P=128
USA Road Network	23,947,347	58,333,344	4.13	3.17	2.64	4.26
RMAT (nasty)	33,554,432	266,636,848	8.60	4.47	2.59	1.89
RMAT	134,217,727	1,071,420,576	59.7	30.7	16.1	8.98
RMAT	134,217,727	2,139,802,777	101.8	52.2	31.6	14.8
RMAT	134,217,727	4,270,508,334	172.7	146.6	125.0	116.3
RMAT	2,147,483,647	17,179,869,184	618.8	314.8	163.1	86.6

sparse, planar graph of the US road network. The rest of the graphs are synthetic RMAT graphs with small-world properties. Using 128 processors, we can determine the connected components of the graph in under 2 minutes.

In [76], MTGL running on a 128-processor Cray XMT computes the connected components of an RMAT graph (with so-called “nasty” parameters) with 33.5 million vertices and an average degree of 8 in approximately 8 seconds. On a generated graph with the same RMAT parameters on the same size machine, GraphCT is able to compute the connected components in 1.89 seconds.

3.2 Clustering Coefficients

Clustering coefficients measure the density of closed triangles in a network and are one method for determining if a graph is a small-world graph [31]. We adopt the terminology of [31] and limit our focus to *undirected* and unweighted graphs. A triplet is an ordered set of three vertices, (i, v, j) , where v is considered the focal point and there are undirected edges $\langle i, v \rangle$ and $\langle v, j \rangle$. An open triplet is defined as three vertices in which only the required two are connected, for example the triplet (m, v, n) shown in Figure 5. A closed triplet is defined as three vertices in which there are three edges, or the triplet (i, v, j) in Figure 5. A triangle is made up of three closed triplets, one for each vertex of the triangle.

The global clustering coefficient C is a single number describing the number of closed triplets over the total number of triplets,

$$C = \frac{\text{number of closed triplets}}{\text{number of triplets}} = \frac{3 \times \text{number of triangles}}{\text{number of triplets}}. \quad (4)$$

The local clustering coefficient C_v is defined similarly for each vertex v ,

$$C_v = \frac{\text{number of closed triplets centered around } v}{\text{number of triplets centered around } v}. \quad (5)$$

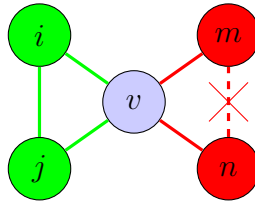


Figure 5: There are two triplets around v in this unweighted, undirected graph. The triplet (m, v, n) is open, there is no edge $\langle m, n \rangle$. The triplet (i, v, j) is closed.

Let e_k be the set of neighbors of vertex k , and let $|e|$ be the size of set e . Also let d_v be the degree of v , or $d_v = |e_v|$. We show how to compute C_v by expressing it as

$$C_v = \frac{\sum_{i \in e_v} |e_i \cap (e_v \setminus \{v\})|}{d_v(d_v - 1)} = \frac{T_v}{d_v(d_v - 1)}. \quad (6)$$

For the remainder of this section, we concentrate on the calculation of local clustering coefficients. Computing the global clustering coefficient requires an additional sum reduction over the numerators and denominators.

Our test data is generated by sampling from a Kronecker product using the RMat recursive matrix generator [73] with probabilities $A = 0.55$, $B = 0.1$, $C = 0.1$, and $D = 0.25$. Each generated graph has a few vertices of high degree and many vertices of low degree. Given the RMat scale k , the number of vertices $n = 2^k$, and an edge factor f , we generate approximately $f \cdot n$ unique edges for our graph.

The clustering coefficients algorithm simply counts all triangles. For each edge $\langle u, v \rangle$, we count the size of the intersection $|e_u \cap e_v|$. The algorithm runs in $O(\sum_v d_v^2)$ time where v ranges across the vertices and the structure is pre-sorted. The multi-threaded implementation also is straightforward; we parallelize over the vertices.

In Figure 6, the scalability of the local clustering coefficients implementation on the Cray XMT is plotted. On an undirected, synthetic RMat graph with approximately 16 million vertices and 135 million edges (left), we are able to calculate all clustering coefficients in 87 minutes on a single processor and 56 seconds on 128 processors. The speedup is 94x. Parallelizing over the vertices, we obtain the best performance when instructing the compiler to schedule the outer loop using futures. The implementation scales almost linearly through 80 processors, then increases more gradually.

In the plot on the right, the same kernel is run on the USA road network, a graph with 24 million vertices and 58 million edges. The graph is nearly planar with a small, uniform degree distribution. Because the amount of work per vertex is nearly equal,

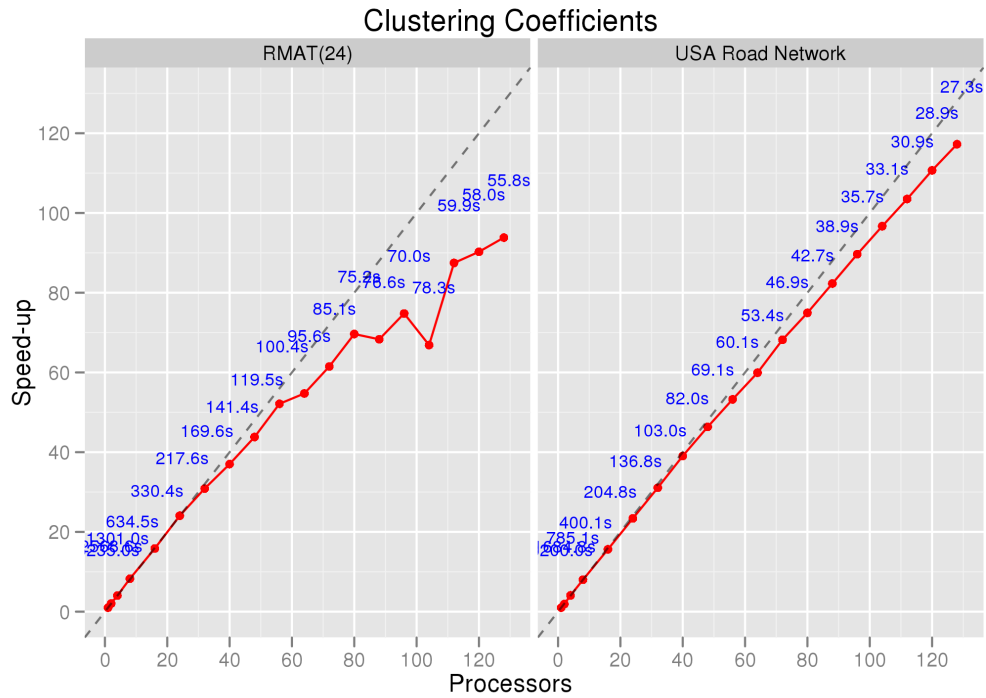


Figure 6: Scalability of the local clustering coefficients kernel on the Cray XMT. On the left, the input graph is an undirected, scale-free graph with approximately 16 million vertices and 135 million edges. The speedup is 94x on 128 processors. On the right, the input graph is the USA road network with 24 million vertices and 58 million edges. The speedup is 120x on 128 processors. Execution times in seconds are shown in blue.

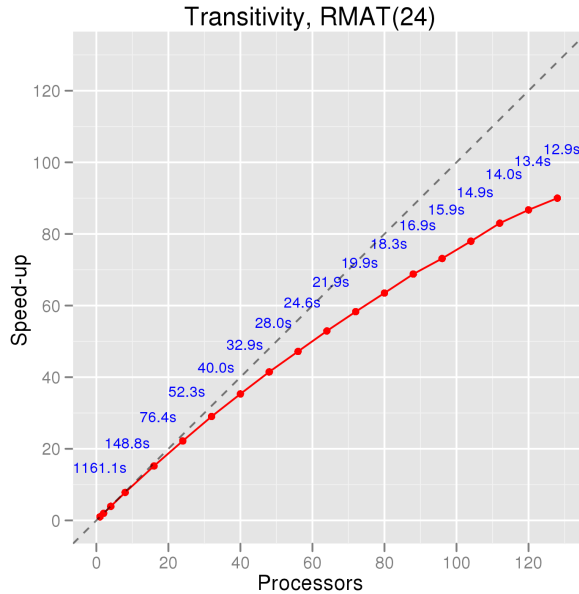


Figure 7: Scalability of the transitivity coefficients kernel on the Cray XMT. The input graph is a directed, scale-free graph with approximately 16 million vertices and 135 million edges. Execution times in seconds are shown in blue. On 128 processors, we achieve a speedup of 90x.

the vertices are easily scheduled and the algorithm scales linearly to 128 processors. The total execution time is about 27 seconds.

These results highlight the challenges of developing scalable algorithms on massive graphs. Where commodity platforms often struggle to obtain a speedup, the latency tolerance and massive multithreading of the Cray XMT enable linear scalability on regular, uniform graphs. The discrepancy between the RMat scalability (left) and the road network (right) is an artifact of the power-law degree distribution of the former. Despite the complex and irregular graph topology, GraphCT is still able to scale up to 128 processors.

There are several variations of clustering coefficients for directed graphs. A straightforward approach is to apply the definition directly and count the number of triangles, where a triangle now requires six edges instead of three. A more sophisticated approach is called the transitivity coefficients. Transitivity coefficients count

the number of transitive triplets in the numerator. A transitive triplet is one in which edges exist from vertex a to vertex b and from vertex b to vertex c , with a shortcut edge from vertex a to vertex c [77].

Transitivity coefficients are implemented in a similar style to clustering coefficients with some differences. We take advantage of the fact that the internal representation of the graph is directed, even for undirected graphs. The directed graph representation provides access to outgoing edges only. To determine incoming edges, we create a second graph data structure that is the undirected version of the directed input graph. Comparing the edge lists for a particular vertex allows efficient calculation of the coefficients.

The scalability of the transitivity coefficients kernel on the Cray XMT is plotted in Figure 7. The input graph is a directed RMAT graph with 16 million vertices and 135 million edges. We do not use loop futures to schedule the outer loop in this case. On a single processor, the calculation requires 20 minutes. On 128 processors, the execution time is under 13 seconds. The speedup is 90x.

3.3 *k*-Betweenness Centrality

The traditional definition of betweenness centrality [8] enumerates all shortest paths in a graph and defines betweenness centrality in terms of the ratio of shortest paths passing through a vertex v . This metric has proven valuable for a number of graph analysis applications, but fails to capture the robustness of a graph. A vertex that lies on a number of paths whose length is just one greater than the shortest path receives no additional value compared to a vertex with an equally large number of shortest paths, but few paths of length one greater.

We will define k -betweenness centrality in the following manner. For an arbitrary graph $G(V, E)$, let $d(s, t)$ denote the length of the shortest path between vertices s and t . We define σ_{st_k} to be the number of paths between s and t whose length is less than or equal to $d(s, t) + k$. Likewise, $\sigma_{st_k}(v)$ is the count of the subset of these paths that pass through vertex v . Therefore, k -betweenness centrality is given by

$$BC_k(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st_k}(v)}{\sigma_{st_k}}. \quad (7)$$

This definition of k -betweenness centrality subsumes Freeman’s definition of betweenness centrality for $k = 0$.

Brandes offered the first algorithm for computing betweenness in $O(mn)$ time for an unweighted graph [78]. Madduri and Bader developed a parallel betweenness algorithm motivated by Brandes’ approach that exploits both coarse- and fine-grained parallelism in low-diameter graphs in [79] and improved the performance of this algorithm using lock-free methods in [38]. Here we extend the latter work to incorporate our new analytic of k -betweenness centrality.

The Cray XMT implementation is similar to that used in previous work [38]. The successor arrays allows us to update δ -values without locking. The optimizations in our code for k -betweenness centrality exploit the fact that we are mostly interested

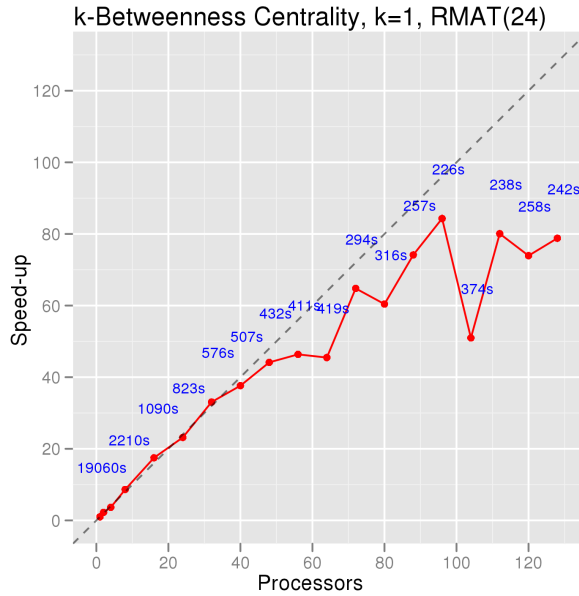


Figure 8: Parallel scaling on the Cray XMT, for a scale-free, undirected graph with approximately 16 million vertices and 135 million edges. Scaling is nearly linear up to 96 processors and speedup is roughly 78 on all 128 processors. $k = 1$ with 256 random sources (single node time 318 minutes). Execution times in seconds shown in blue.

in small values of k . There are several nested loops, however most of them are very simple for small k and unfurl quickly. By manually coding these loops for smaller values of k' , we significantly reduce the execution time, since the time to set up and iterate over the small number of loop iterations quickly outstrips the actual useful work inside of them. For an undirected, scale-free graph with 1 million vertices and 8 million edges, the time to compute 1-betweenness centrality drops by a factor of two with this optimization.

In Figure 8, we show the parallel scaling of our optimized code on the 128-processor Cray XMT. We reduced the execution time from several hours down to a few minutes for this problem. To accommodate the more than 12,000 hardware thread contexts

available on this machine, we run multiple breadth-first searches in parallel and instruct the compiler to schedule main loop iterations using *loop futures*. Each breadth-first search is dependent upon a single vertex queue that is accessed atomically and quickly becomes a hotspot. By doing so, however, the memory footprint is multiplied by the number of concurrent searches. On 128 processors, a graph with 135 million edges takes about 226 seconds to run for $k = 1$ *approximate* betweenness. This approximation is based on selecting a random sample of source vertices s . For these experiments, the number of starting vertices is 256. The plot shows good scaling up to our machine size.

Evaluating k -Betweenness

To explore the effect of various values of k on the calculation of k -betweenness centrality, we apply our Cray XMT implementation to the ND-www graph data set [80]. This graph represents the hyperlink connections of web pages on the Internet. It is a directed graph with 325,729 vertices and 1,497,135 edges. Its structure demonstrates a power-law distribution in the number of neighbors. The graph displays characteristics typical of scale-free graphs found in social networks, biological networks, and computer networks.

To examine the graph data, we compute k -betweenness centrality for k from 0 (traditional betweenness centrality) to 2. The betweenness scores are compared for each value of k . An analysis directly follows. Also, after computing betweenness centrality for $k = 0$, we remove one or more of the highest ranking vertices and re-examine the results.

Looking at the highest ranking vertices going from $k = 0$ to $k = 2$, the subset of vertices and the relative rankings change little. This seems to indicate that the paths k longer than the shortest path lie along the same vertices as the shortest paths in this graph. As predicted, the traditional betweenness centrality metric fails to capture all of the information in the graph. When examining the BC_k score for $k > 0$ of vertices

Table 3: The number of vertices ranked in selected percentiles for $k = 1$ and $k = 2$ whose betweenness centrality score was 0 for $k = 0$ (traditional BC). There were 14,320 vertices whose traditional BC score was 0, but whose BC_k score for $k = 1$ was greater than 0. The ND-www graph contains 325,729 vertices and 1,497,135 edges.

Percentile	$k = 1$	$k = 2$
90th	513	683
95th	96	142
99th	11	12

whose score for $k = 0$ was 0 (no shortest paths pass through these vertices), it is clear that a number of very important vertices in the graph are not counted in traditional betweenness centrality. For $k = 1$, 417 vertices are ranked in the top 10 percent, but received a score of 0 for $k = 0$. In the 99th percentile are 11 vertices. Likewise, 12 vertices received a traditional BC score of 0, but ranked in the top 1 percent for $k = 2$. There are 14,320 vertices whose betweenness centrality score for $k = 0$ was 0, but had a k -betweenness centrality score of greater than 0 for $k = 1$ (Figure 9).

Given that the execution time of this algorithm grows exponentially with k , it is desirable to understand the effect of choosing a given value for k . In Figure 10, we see that increasing k from 0 to 1 captures significantly more path data. However, increasing from $k = 1$ to $k = 2$ displays much less change. It is reasonable to believe that small values of k for some applications may capture an adequate amount of information while remaining computable.

The vertices that get overlooked by traditional betweenness centrality, but are captured by k -betweenness centrality, play an important role in the network. They do not lie along any shortest paths, but they lie along paths that are very close to the shortest path. If an edge is removed that breaks one or more shortest paths, these vertices would likely become very central to the graph. The traditional definition of betweenness centrality fails to capture this subtle importance, but k -betweenness centrality is more robust to noisy data and makes it possible to identify these vertices.

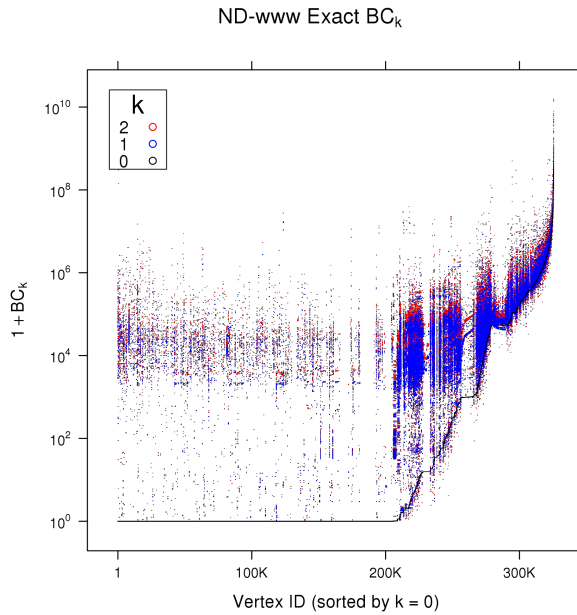


Figure 9: Per-vertex betweenness centrality scores for $k = 0, 1,$ and $2,$ sorted in ascending order for $k = 0.$ Note the number of vertices whose score is several orders of magnitude larger for $k = 1$ or 2 than for traditional betweenness centrality.

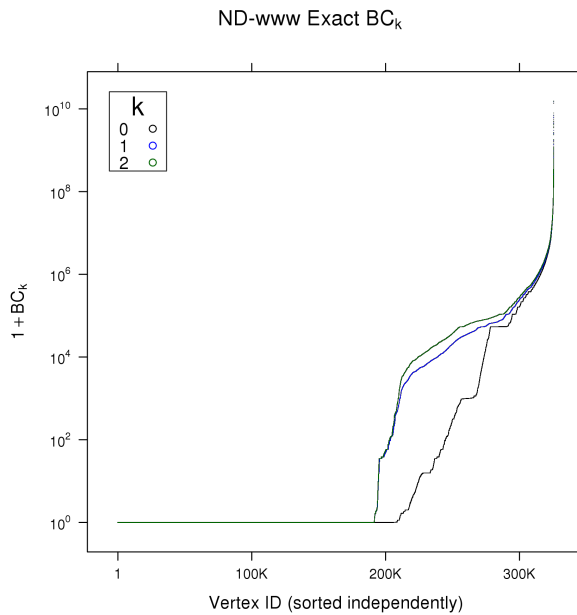


Figure 10: Per-vertex betweenness centrality scores for $k = 0, 1,$ and $2,$ sorted independently in ascending order for each value of $k.$

When a vertex of high betweenness is removed from the graph, it causes a number of changes in betweenness scores for all values of k that we are studying. Many vertices gain a small number of shortest paths and their ranking is fairly unchanged. In general, those vertices ranked highest on the list remain at the top. This would seem to indicate that there is a network of short paths between vertices of extremely high betweenness. Interestingly, however, other vertices jump wildly within the rankings. Often, several of these are neighbors of the removed vertex. This underscores the previous conclusion that a vertex of relatively little importance in the graph can become extremely important if the right vertex or combination of vertices are removed.

CHAPTER IV

STINGER

The growth of social media, heightened interest in knowledge discovery, and the rise of ubiquitous computing in mobile devices and sensor networks [59] have motivated researchers and domain scientists to ask complex queries about the massive quantity of data being produced. During a recent Champions League football match between Barcelona and Chelsea, Twitter processed a record 13,684 Tweets per second [81]. Facebook users posted an average of 37,000 Likes and Comments per second during the first quarter of 2012 [82]. Google’s Knowledge Graph for search clustering and optimization contains 500 million objects and 3.5 billion relationships [83].

In the *massive streaming data analytics* model, we view the graph as an infinite stream of edge insertions, deletions, and updates. Keeping complex analytics up to date at these high rates is a challenge that requires new algorithms that exploit opportunities for partial recomputation, new data structures that maximize parallelism and reduce locking, and massive multithreaded compute platforms. In most cases, the new information being ingested does not affect the entire graph, but only a small neighborhood around the update. Rather than recomputing an analytic from scratch, it is possible to react faster by only computing on the data that have changed. Algorithms that take advantage of this framework need a flexible, dynamic data structure that can tolerate the ingest rate of new information.

STINGER (Spatio-Temporal Interaction Networks and Graphs Extensible Representation) is a high performance, extensible data structure for dynamic graph problems [30]. The data structure is based on linked lists of blocks. The number of vertices and edges can grow over time by adding additional vertex and edge blocks.

Both vertices and edges have types, and a vertex can have incident edges of multiple types.

Edges incident on a given vertex are stored in a linked list of edge blocks. An edge is represented as a tuple of neighbor vertex ID, type, weight, and two timestamps. All edges in a given block have the same edge type. The block contains metadata such as the lowest and highest timestamps and the high-water mark of valid edges within the block.

Parallelism exists at many levels of the data structure. Each vertex has its own linked list of edge blocks that is accessed from the logical vertex array (LVA). A “for all vertices” loop is parallelized over these lists. Within an edge block, the incident edges can be explored in a parallel loop. The size of the edge block, and therefore the quantity of parallel work to be done, is a user-defined parameter. In our experiments, we arbitrarily set the edge block size to 32.

The edge type array (ETA) is a secondary index that points to all edge blocks of a given type. In an algorithm such as connected components that is edge parallel, this additional mode of access into the data structure permits all edge blocks to be explored in a parallel for loop.

To assist the programmer in writing a graph traversal, our implementation of STINGER provides parallel edge traversal macros that abstract the complexities of the data structure while still allowing compiler optimization. For example, the `STINGER_PARALLEL_FORALL_EDGES_OF_VTX` macro takes a STINGER data structure pointer and a vertex ID. The programmer writes the inner loop as if he or she is looking at a single edge. Edge data is read using macros such as `STINGER_EDGE_TYPE` and `STINGER_EDGE_WEIGHT`. More complex traversal macros are also available that filter the edges seen based on timestamp and edge type.

The STINGER specification does not specify consistency; the programmer must assume that the graph can change underneath the application. The programmer is

provided routines to extract a snapshot of a vertex’s neighbor list, alleviating this concern at the expense of an additional buffer in memory.

Although most analytic kernels will only read from the data structure, the STINGER must be able to respond to new and updated edges. Functions are provided that insert, remove, increment, and touch edges in parallel. The graph can be queried as to the in-degree and out-degree of a vertex, as well as the total number of vertices and edges in the graph.

STINGER is written in C with OpenMP and Cray MTA pragmas for parallelization. It compiles and runs on both Intel and AMD x86 platforms and the Cray XMT supercomputing platform, with experimental support for Python and Java on x86 systems. The code is available under BSD license at <http://www.cc.gatech.edu/stinger>.

4.1 Experimental Setup

We will examine STINGER implementations and performance on two multithreaded systems with large-scale memories. The first is a 4-socket Intel multicore system (mirasol) employing the Intel Xeon E7-8870 processor at 2.40 GHz with 30 MiB of L3 cache per processor. Each processor has 10 physical cores and supports Hyper-Threading for a total of 80 logical cores. The server is equipped with 256 GiB of 1066 MHz DDR3 DRAM.

The second system is the Cray XMT (and the next generation Cray XMT2) [32]. The Cray XMT is a massively multithreaded, shared memory supercomputer designed specifically for graph problems. Each processor contains 128 hardware streams and can execute a different stream at each clock tick. Low-overhead synchronization is provided through atomic fetch-and-add operations and full-empty bit memory semantics. Combined, these features enable applications with large quantities of parallelism to overcome the long latency of irregular memory access. The Cray XMT system at

Pacific Northwest National Lab (`cougarxmt`) has 128 Threadstorm processors with 1 TiB main memory. The Cray XMT2 system at the Swiss National Supercomputing Centre (`matterhorn`) has 64 processors and 2 TiB main memory.

Due to a variety of concerns, e.g. privacy, company proprietary and data set size, it is often difficult to obtain data sets from social media and other sources at the scale of billions of edges. We substitute synthetic graphs to approximate the behavior of our algorithms at scale. For these experiments, we utilize the popular RMAT [73] synthetic graph generator, which produces scale-free graphs with a power-law distribution in the number of neighbors.

Our experiments begin with an initial graph in memory from the RMAT generator (we use RMAT parameters $a = 0.55, b = 0.1, c = 0.1, d = 0.25$). The graph size is given by two parameters: *scale* and *edgefactor*. The initial graph has 2^{scale} vertices and approximately $2^{scale} * edgefactor$ edges. After generation, we make the graph undirected.

After generating the initial graph, we generate additional edges – using the same generator and parameters – to form a stream of updates. This stream of updates is mostly edge insertions. With a probability of 6.25 percent, we select some of these edge insertions to be placed in a deletion queue. With the same probability, we take an edge from the deletion queue and add it to the stream as an edge deletion.

The insert/remove microbenchmark builds a STINGER data structure in memory from the generated initial graph on disk. Next, a batch of edge updates is taken from the generated edge stream. The number of edge updates in the batch is variable. We measure the time taken to process each edge update in the data structure. We measure several batches and report the performance in terms of updates per second.

4.2 *Optimizations*

Applications that rely on STINGER typically receive a constant stream of new edges and edge updates. The ability to react quickly to new edge information is a core feature of STINGER. When an update on edge $\langle u, v \rangle$ is received, we must first search all of the edge blocks of vertex u for neighbor v of the given edge type. If the edge is found, the weight and timestamp are updated accordingly. If the edge is not found, an empty space must be located or an empty edge block added to the list.

In an early implementation of STINGER, each new edge was processed in this manner one at a time. This approach maximized our ability to react to a single edge change. On an Intel multicore system with a power-law graph containing 270 million edges, inserting or updating one at a time yielded a processing rate of about 12,000 updates per second, while the Cray XMT achieved approximately 950 updates per second. The Cray XMT performance is low because single edge updates lack concurrency required to achieve high performance.

On systems with many thread contexts and memory banks, there is often insufficient work or parallelism in the data structure to process a single update at a time. To remedy this problem, we began processing edge updates in batches. A batch amortizes the cost of entering the data structure and provides a larger quantity of independent work to do.

A later implementation of STINGER first sorts the batch (typically 100,000 edge updates at a time) such that all edge updates incident on a particular vertex are grouped together with deletions separated from insertions. For each unique vertex in the batch, we have at least one work item that can be performed in parallel. Deletions are processed prior to insertions to potentially make room for the new edges. Updates on a particular vertex are done sequentially to avoid synchronization.

This approach to updates yields a 14x increase on the Intel multicore system. We can process 168,000 updates per second. The Cray XMT implementation reaches

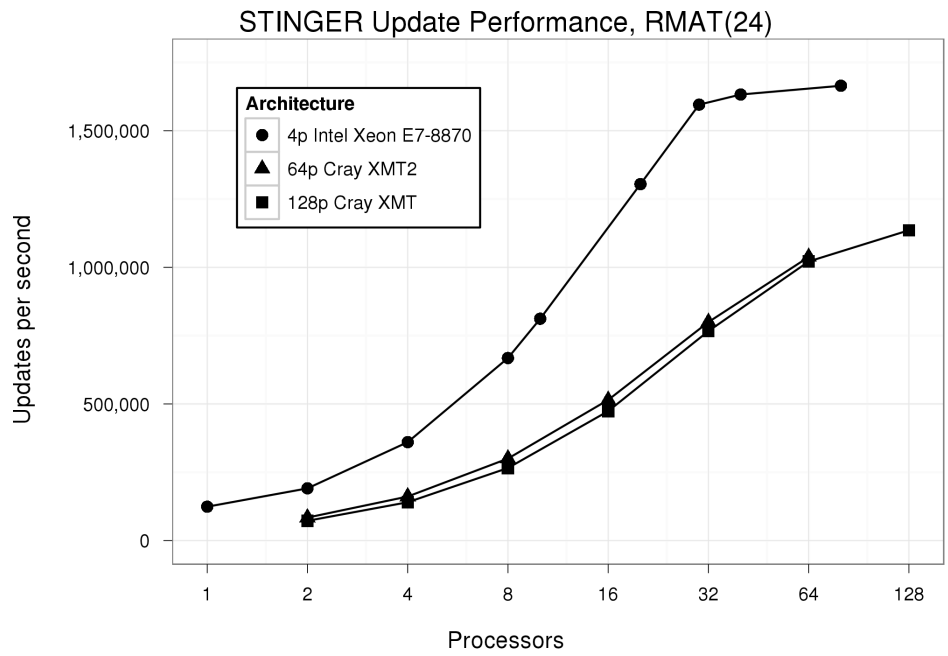


Figure 11: Updates per second on a scale-free graph with approximately 16 million vertices and 270 million edges and a batch size of 100,000 edge updates.

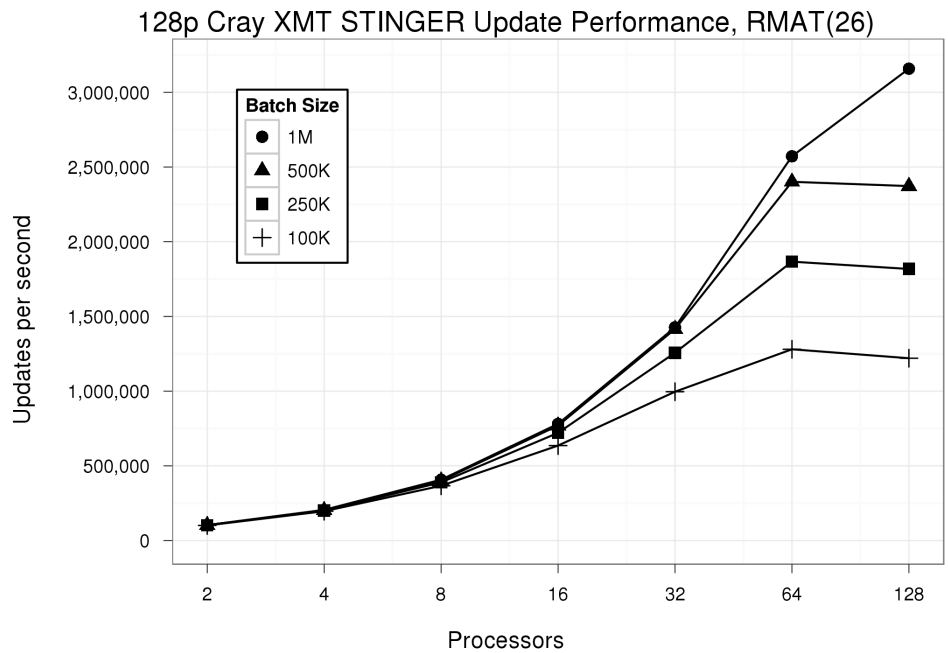


Figure 12: Increasing batch size results in better performance on the 128-processor Cray XMT. The initial graph is a scale-free graph with approximately 67 million vertices and 537 million edges.

225,000 updates per second, or a 235x increase in performance.

In a scale-free graph, however, a small number of vertices will face many updates, while most will only have a single update or no updates at all. This workload imbalance limits the quantity of parallelism we can exploit and forces most threads to wait on a small number of threads to complete.

To solve this problem, we skip sorting the edges and process each edge insertion independently and in parallel. However, processing two edge updates incident on the same vertex introduces race conditions that must be handled with proper synchronization. The Cray XMT is a perfect system for this scenario. The additional parallelism will increase machine utilization and its fine-grained synchronization intrinsics will enable a simple implementation.

There are three possible scenarios when inserting an edge into a vertex's adjacency list in STINGER. If the edge already exists, the insert function should increment the edge weight and update the modified timestamp. If the edge does not exist, a new edge should be inserted in the first empty space in an edge block of the appropriate type. If there are no empty spaces, a new edge block containing the new edge should be allocated and added to the list.

The parallel implementation guarantees these outcomes by following a simple protocol using full-empty semantics on the Cray XMT or using an emulation of full-empty semantics built on atomic compare-and-swap instructions on x86. Since multiple threads reading and writing in the same place in an adjacency list is a relatively uncommon occurrence, locking does not drastically limit performance. When an edge is inserted, the linked list of edge blocks is first checked for an existing edge. If the edge is found, the weight is incremented atomically. Otherwise the function searches the linked list a second time looking for an empty space. If one is found, the edge weight is locked. Locking weights was chosen to allow readers within the first search

to continue past the edge without waiting. If the edge space is still empty after acquiring the lock, the new edge is written into the block and the weight is unlocked. If the space is not empty but has been filled with the same destination vertex as the edge being inserted, the weight is incremented and the weight is unlocked. If another edge has been written into the space before the lock was acquired, the weight is unlocked and the search continues as before. If the second search reaches the end of the list having found no spaces, the “next block” pointer on the last edge block must be locked. Once it is locked it is checked to see if the pointer is still null indicating the end of the list. If so, a new block is allocated and the edge is inserted into it before linking the block into the list and unlocking the previous “next” pointer. If the pointer is not null, it is unlocked, and the search continues into the next block. In this way, we guarantee that all insertions are successful, that all destination vertices are unique, that no empty space is wasted, and that no new blocks are needlessly allocated. Deletions are handled by a similar algorithm.

Implemented in this way, the Cray XMT reaches 1.14 million updates per second on the scale-free graph with 270 million edges. This rate is 1,200x faster than the single update at a time approach. With this approach, we also have sufficient parallelism such that the performance scales to our full system of 128 processors. The performance of the Cray XMT, Cray XMT2, and an Intel multicore system on the same problem is compared in Figure 11.

On the 4-socket Intel multicore system, this method achieves over 1.6 million updates per second on the same graph with a batch size of 100,000 and 1.8 million updates per second with a batch size of 1,000,000. This is 133x faster than the single update at a time approach and nearly 10x faster than the batch sorting approach. The scalability of this approach is linear to 20 threads, but falls off beyond that mark due to limitations imposed by the use of atomics across multiple sockets.

While the Intel system performs well, the problem size is constrained by memory.

As we increase the scale of the problem, only the Cray XMT can accommodate the larger problem sizes. Hence, Figure 12 only includes Cray XMT results.

With a larger graph (67 million vertices and 537 million edges), the performance remains flat at 1.22 million updates per second. Increasing the batch size from 100,000 updates to one million updates further increases the available parallelism. In Figure 12, the increased parallelism from increasing batch sizes results in better scalability. The Cray XMT reaches a peak of 3.16 million updates per second on 128 processors for this graph.

The Cray XMT2, which has a 4x higher memory density than the Cray XMT, can process batches of one million updates on a scale-free graph with 268 million vertices and 2.15 billion edges at 2.23 million updates per second. This quantity represents a 44.3x speedup on 64 processors over a single processor. The graph in memory consumes approximately 313 GiB.

4.3 Streaming Clustering Coefficients

Clustering coefficients measure the density of closed triangles in a network and are one method for determining if a graph is a small-world graph [31]. We adopt the terminology of [31] and limit our focus to *undirected* and unweighted graphs. A triplet is an ordered set of three vertices, (i, v, j) , where v is considered the focal point and there are undirected edges $\langle i, v \rangle$ and $\langle v, j \rangle$. An open triplet is defined as three vertices in which only the required two are connected, for example the triplet (m, v, n) in Figure 13. A closed triplet is defined as three vertices in which there are three edges, or triplet (i, v, j) in Figure 13. A triangle is made up of three closed triplets, one for each vertex of the triangle.

The global clustering coefficient C is a single number describing the number of closed triplets over the total number of triplets,

$$C = \frac{\text{number of closed triplets}}{\text{number of triplets}} = \frac{3 \times \text{number of triangles}}{\text{number of triplets}}. \quad (8)$$

The local clustering coefficient C_v is defined similarly for each vertex v ,

$$C_v = \frac{\text{number of closed triplets centered around } v}{\text{number of triplets centered around } v}. \quad (9)$$

Let e_k be the set of neighbors of vertex k , and let $|e|$ be the size of set e . Also let d_v be the degree of v , or $d_v = |e_v|$. We show how to compute C_v by expressing it as

$$C_v = \frac{\sum_{i \in e_v} |e_i \cap (e_v \setminus \{v\})|}{d_v(d_v - 1)} = \frac{T_v}{d_v(d_v - 1)}. \quad (10)$$

To update C_v as edges are inserted and deleted, we maintain the degree d_v and the triangle count T_v separately.

For the remainder of the discussion, we concentrate on the calculation of local clustering coefficients. Computing the global clustering coefficient requires an additional sum reduction over the numerators and denominators.

An inserted edge increments the degree of each adjacent vertex, and a deleted edge decrements the degrees. Updating the triangle count T_v is more complicated. Algorithm 2 provides the general framework. Acting on edge $\langle u, v \rangle$ affects the degrees only of u and v but may affect the triangle counts of all neighbors. With atomic increment operations available on most high-performance platforms, all of Algorithm 2's

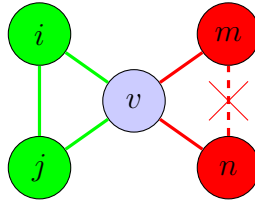


Figure 13: There are two triplets around v in this unweighted, undirected graph. The triplet (m, v, n) is open, there is no edge $\langle m, n \rangle$. The triplet (i, v, j) is closed.

Algorithm 2 An algorithmic framework for updating local clustering coefficients. All loops can use atomic increment and decrement instructions to decouple iterations.

Input: Edge $\langle u, v \rangle$ to be inserted (+) or deleted (-), local clustering coefficient numerators T , and degrees d

Output: Updated local triangle counts T and degrees d

```
1:  $d_u \leftarrow d_u \pm 1$ 
2:  $d_v \leftarrow d_v \pm 1$ 
3:  $count \leftarrow 0$ 
4: for all  $x \in e_v$  do
5:   if  $x \in e_u$  then
6:      $T_x \leftarrow T_x \pm 1$ 
7:      $count \leftarrow count \pm 1$ 
8:  $T_u \leftarrow T_u \pm count$ 
9:  $T_v \leftarrow T_v \pm count$ 
```

loops can be parallelized fully.

The search in line 5 can be implemented in several ways. A brute-force method simply iterates over every element in e_v for each x , explicitly searching for all new closed triplets given a new edge $\langle u, v \rangle$. The running time of the algorithm is $O(d_u d_v)$, which may be problematic when two high-degree vertices are affected.

If the edge list is kept sorted as in a static computation, the intersection could be computed more efficiently in $O(d_u + d_v)$ time. However, the cost of keeping our dynamic data structure sorted outweighs the update cost. We can, however, accelerate the method to $O((d_u + d_v) \log d_u)$ by sorting the current edge list of d_v and searching for neighbors with bisection. The sorting routine can employ a parallel sort, and iterations of the search loop can be run in parallel given atomic addition / subtraction operations.

We present a novel set intersection approximation algorithm with constant-time search and query properties and an extremely high degree of accuracy. We summarize neighbor lists with Bloom filters [84], a probabilistic data structure that gives false positives (but never false negatives) with some known probability. We then query

against this Bloom filter to determine if the intersection exists.

Edge adjacencies could be represented as bit arrays. In one extreme, each neighbor list could be an array using one bit per vertex as well as an edge list. Then $|e_u \cap e_v|$ can be computed in $O(\min\{d_u, d_v\})$ time by iterating over the shorter edge list and checking the bit array. However, maintaining $O(n)$ storage per source vertex is infeasible for massive graphs.

Instead, we approximate an edge list by inserting its vertices into a Bloom filter. A Bloom filter is also a bit array but uses an arbitrary, smaller number of bits. Each edge list e_v is summarized with a Bloom filter for v . A hash function maps a vertex $w \in e_v$ to a specific bit in this much smaller array. With fewer bits, there may be hash collisions where multiple vertices are mapped to the same bit. These will result in an overestimate of the number of intersections.

A Bloom filter attempts to reduce the occurrence of collisions by using k independent hash functions for each entry. When an entry is inserted into the filter, the output of the k hash functions determines k bits to be set in the filter. When querying the filter to determine if an edge exists, the same k hash functions are used and each bit place is checked. If any bit is set to 0, the edge cannot exist. If all bits are set to 1, the edge exists with a high probability.

Bloom filters have several parameters useful to fix a given probability of failure. In-depth description of Bloom filters' theory is beyond the scope of this paper, but a few useful features include the following: Bloom filters never yield false negatives where an edge is ignored, only false positives where a non-existent edge is counted. The probability of falsely returning membership is approximately $(1 - e^{-kd_u/m})^k$ where m is the length of the filter. This can be optimized by setting k to an integer near $\ln 2 \times m/d$ [85], choosing d according to the expected degrees in the graph. Our initial implementation uses two hash functions, $k = 2$, and a 1 MiB filter. The probability of a false-positive will vary depending on the degree of the vertex. In a scale-free

graph with an average degree of 30 and a maximum degree of 200,000, the average false-positive rate will be 5×10^{-11} and the worst-case rate will be 2×10^{-3} .

When intersecting two high-degree vertices, the Bloom filter holds a slight asymptotic advantage over sorting one edge list, but a multithreaded implementation benefits from additional parallelism throughout the Bloom filter’s operations. Note that entries cannot be deleted from a Bloom filter. A new filter is constructed for each application of Algorithm 2, so we never need to delete entries.

Modifications to Algorithm 2 for supporting a Bloom filter are straight-forward. After line 3, initialize the Bloom filter using vertices in e_u :

- 1: **for all** $y \in e_u$ **do**
- 2: **for** $i = 1 \rightarrow k$ **do**
- 3: Set bit $H_i(y)$ in B_x to 1

Then implement the search in line 5 as follows:

- 1: **for** $i = 1 \rightarrow k$ **do**
- 2: **if** bit $H_i(x) = 0$ **then**
- 3: Skip to next x

Local clustering coefficients’ properties help us batch the input data. Recomputing changed coefficients only at the end of the batch’s edge actions frees us to reorder the insertions and deletions. Reordering repeated insertions and removals of the same edge may alter the edge’s auxiliary data, however, so we must take some care to resolve those in sequence order. After resolving actions on the same edge, we process all removals before all insertions to open edge slots and delay memory allocation.

The batch algorithm is as follows:

- 1: Transform undirected edges $\langle i, j \rangle$ into pairs of directed edges $i \rightarrow j$ and $j \rightarrow i$ because STINGER stores directed edges.
- 2: Group edges within the batch by their source vertex.

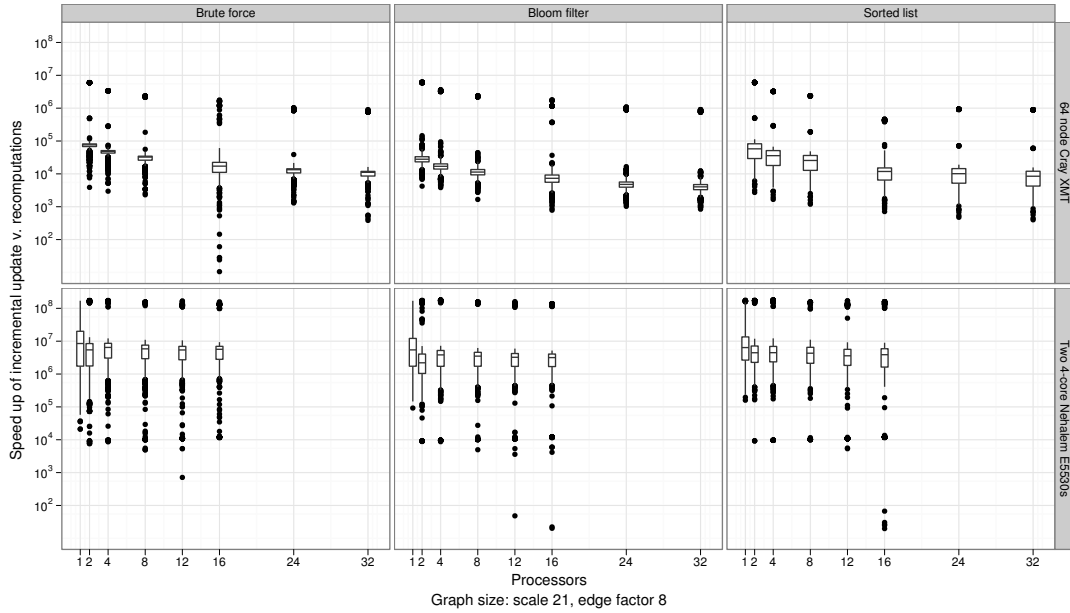


Figure 14: Speedup of incremental, local updates relative to recomputing over the entire graph.

- 3: Resolve operations on the same edge in sequence order.
- 4: Apply all removals, then all insertions to the STINGER structure.
- 5: Recompute the triangle counts and record which vertices are affected.
- 6: Recompute the local clustering coefficients of the affected vertices.

In step 5, we use slight variations of the previous algorithms. The source vertex’s neighboring vertices are gathered only once, and the array is re-used across the inner loop. The sorted update and Bloom filter update compute their summary data using the source vertex rather than choosing the larger list.

Unlike calculating the triangle counts T for the entire graph, updating T for an individual edge insertion or deletion exposes a variable amount of fine-grained parallelism. We present results showing how aggregate performance of a *single* edge insertion or deletion stays relatively constant.

The sequential complexity of our update algorithms is summarized in Table 4. Boxplots summarizing the updates per second achieved on our test platforms are

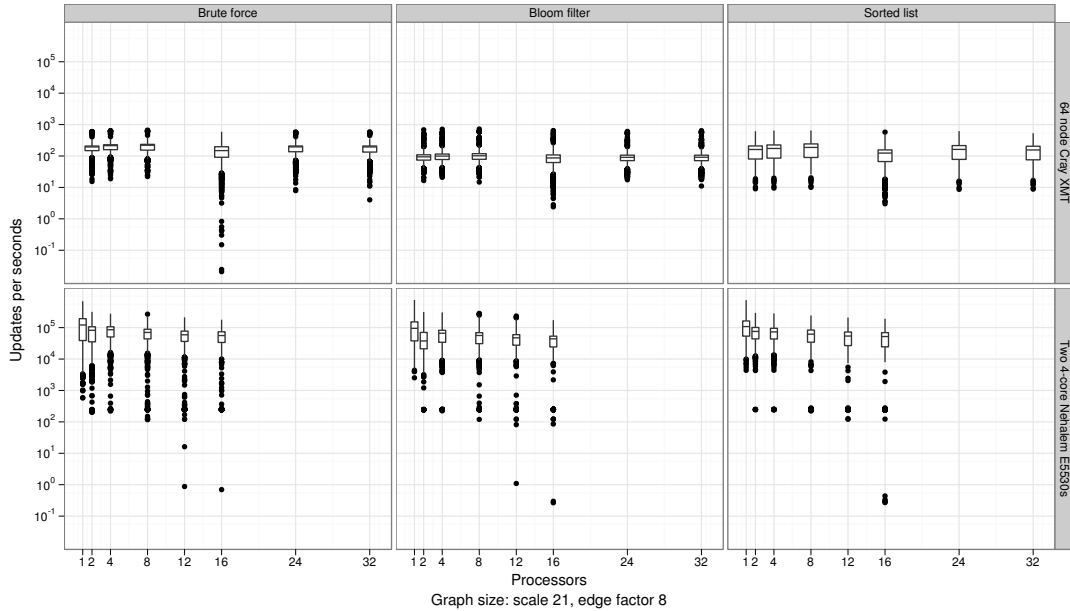


Figure 15: Updates per second by algorithm.

Table 4: Summary of update algorithms

Algorithm	Update Complexity
Brute force	$O(d_u d_v)$
Sorted list	$O((d_u + d_v) \log d_u)$, $d_u < d_v$
Bloom filter	$O(d_u + d_v)$

presented in Figure 15. Figure 14 shows local recomputation’s speedup of locally relative to globally recomputing the graph’s clustering coefficients. Boxes in Figures 15 and 14 span the 25% – 75% quartiles of the update times. The bar through the box shows the median. The lines stretch to the farthest non-outlier, those within $1.5\times$ the distance between the median and the closest box side. Points are outliers.

In Figure 15, we see the Cray XMT keeps a steady update rate on this relatively small problem regardless of the number of processors. The outliers with 16 processors are a result of sharing resources with other users. The Bloom filter shows the least variability in performance. Figure 14 shows that local recomputation accelerates the update rate typically by a factor of over a thousand.

Table 5: Comparison of single edge versus batched edge updates on 32 Cray XMT processors, in updates per second, on a scale-free graph with approximately 16 million vertices and 135 million edges.

Batch Size	Brute force	Bloom filter
Edge by edge	90	60
Batch of 1000	25,100	83,700
Batch of 4000	50,100	193,300

The Intel Nehalem results degrade with additional threads. The noise at 12 and 16 threads results from over-allocation and scheduling from hyperthreading. The Intel Nehalem outperforms the Cray XMT by several orders of magnitude, but can only hold a graph of approximately 2 million vertices. This Cray XMT is capable of holding a graph in memory up to 135 million vertices.

Table 5 shows performance obtained from batching operations and extracting parallelism on the Cray XMT. The sorting algorithm was not considered for batching. Notice that increasing the batch size greatly improves performance. For the Bloom filter, this comes at the cost of a proportional increase in memory footprint. A batch size of 4000 required choosing a filter size of 1 MiB to fit within the system’s available memory. Even so, we encountered no false positives over 1 million edge actions. Increasing the batch size intuitively improves scalability since data parallelism is increased in the update phase.

4.4 *Streaming Connected Components*

To motivate our approach, we would like to answer, in a persistent manner, the question “do vertex u and vertex v lie in the same component?” while at the same time supporting the ability to insert and delete edges presented as a batch in parallel. We focus our efforts on scale-free networks like those of social networks and other biological networks and capitalize on their structural properties.

Unlike prior approaches, our new framework manufactures large amounts of parallelism by considering the input data stream in batches. All of the edge actions within the batch are processed in parallel. Applying any of the breadth-first search-based algorithms to a batch of deletions would result in thousands of concurrent graph traversals, exhausting the memory and computational resources of the machine.

In this section, we present our algorithmic approach for tracking connected components given a stream of edge insertions and deletions. Optimizations afforded by the scale-free nature of the graph are also highlighted. We take advantage of the low diameter by intersecting neighborhood bit arrays to quickly re-establish connectivity using triangles. Also, our algorithm uses a breadth-first search of the significantly smaller component graph, whose size is limited by number of insertions and deletions in the batch being processed. The scale-free nature of the input data means that most edges in the batch do not span components, so the number of non-self-edges in the component graph for a given batch is very small.

Our techniques fully utilize the fine-grained synchronization primitives of the Cray XMT to look for triangles in the edge deletions and quickly rule out most of the deletions as inconsequential without performing a single breadth-first search. All of the neighbor intersections can be computed concurrently, providing sufficient parallelism for the architecture.

Despite the challenges posed by the input data, we show that the scale-free structure of social networks can be exploited to accelerate graph connectivity queries in light of a stream of deletions. On a synthetic social network with over 16 million vertices and 135 million edges, we are able to maintain persistent queries about the connected components of the graph with an input rate of 240,000 updates per second, a three-fold increase over previous methods.

Hence, our new framework for incremental computation, rather than recomputation, enables scaling to very large data sizes on massively parallel, multithreaded

supercomputing architectures. Note that in this section we add the monitoring of a global property of the data, whereas the previous section was concerned with local clustering coefficients, which are easier to handle due to their locality. As a next step, studying massive dynamic graphs with more complex methods based on our algorithmic kernels will lead to insights about community and anomaly detection unavailable within smaller samples.

Given an edge to be inserted into a graph and an existing labeling of the connected components, one can quickly determine if it has joined two components. Given a deletion, however, recomputation (through breadth-first search or *st*-connectivity) is the only known method with subquadratic space complexity to determine if the deleted edge has cleaved one component into two. If the number of deletions that actually cause structural change is very small compared to the total number of deletions (such as in the case of a scale-free network), our goal will be to quickly rule out those deletions that are “safe” (*i.e.* provably do not split a component). The framework we propose for this computation will establish a threshold for the number of deletions we have not ruled out between recomputations.

Our approach for tracking components is motivated by several assumptions about the input data stream. First, a very small subset of the vertices are involved in any series of insertions and deletions. Insertions that alter the number of components will usually join a small component to the large component. Likewise, a deletion that affects the structure of the graph typically cleaves off a relatively small number of vertices. We do not anticipate seeing the big component split into two large components. The small diameter implies that connectivity between two vertices can be established in a small number of hops, but the low diameter and power-law distribution in the number of neighbors also implies that a breadth-first search quickly consumes all vertices in the component.

Second, we adopt the *massive streaming data analytics* model [28]. We assume

that the incoming data stream of edge insertions and deletions is infinite, with no start or end. We store as much of the recent graph topology as can be held in-memory alongside the data structures for the computation. The graph contains inherent error arising from the noisy data of a social network containing false edges as well as missing edges. As a result, we will allow a small amount of error at times during the computation, so long as we can guarantee correct results at specific points. The interval between these points will depend on the tolerance for error. We process the incoming data stream as batches of edge insertions and deletions, but do not go back and replay the data stream.

The pseudocode of our algorithm appears in Algorithm 3. The algorithm consists of four phases that are executed for each batch of edge insertions and deletions that is received. These phases can be summarized as follows: First, the batch of edges is sorted by source and destination vertices. Second, the edges are inserted and/or deleted in the STINGER data structure. Third, edge deletions are evaluated for their effect on connectivity. Finally, insertions are processed and the affected components are merged.

We will consider unweighted, undirected graphs, as social networks generally require links to be mutual. The graph data structure, the batch of incoming edges currently being processed, and the metadata used to answer queries fit completely within the memory. We can make this assumption in light of the fact that current high-end computing platforms, like the Cray XMT, provide shared memories on the order of terabytes. We will now examine, in detail, each phase of the algorithm.

In the sort phase, we are given a batch of edges to be inserted and deleted. In our experiments, the size of this batch may range from 1,000 to 1 million edges. We use a negative vertex ID to indicate a deletion. The batch must first be sorted by source vertex and then by destination vertex. On the Cray XMT, we bucket sort by source using atomic fetch-and-add to determine the size of the buckets. Within

Algorithm 3 A parallel algorithm for tracking connected components.

Input: Batch B of edges $\langle u, v \rangle$ to be inserted and deleted, component membership M , threshold R_{thresh} , number of relevant deletions R , bit array A , component graph C

Output: Updated component membership M'

```

1: Sort( $B$ )                                     ▷ Phase 1: Prepare batch
2: for all  $b \in B$  in parallel do
3:   if  $b$  is deletion then
4:     Push( $Q_{\text{del}}, b$ )
5:   else
6:     Push( $Q_{\text{ins}}, b$ )
7: StingerDeleteAndInsertEdges( $B$ )             ▷ Phase 2: Update data structure
8: for all  $b \in Q_{\text{del}}$  in parallel do         ▷ Phase 3: Process deletions
9:    $\langle u, v \rangle \leftarrow b$ 
10:   $A_u \leftarrow \vec{0}$                        ▷ All bits set to zero
11:  for all  $n \in \text{Neighbors}(u)$  in parallel do
12:    Set bit  $n$  in  $A_u$  to 1
13:   $F \leftarrow 0$ 
14:  for all  $n \in \text{Neighbors}(v)$  in parallel do
15:    if bit  $n$  in  $A_u = 1$  then
16:       $F \leftarrow 1$                            ▷ Triangle found
17:  if  $F = 0$  then
18:    atomic  $R \leftarrow R + 1$                  ▷ No triangles found
19: if  $R > R_{\text{thresh}}$  then
20:   $R \leftarrow 0$ 
21:   $M' \leftarrow \text{ConnectedComponents}(G)$ 
22: else                                         ▷ Phase 4: Process insertions
23:   $C \leftarrow \emptyset$ 
24:  for all  $b \in Q_{\text{ins}}$  in parallel do
25:     $\langle u, v \rangle \leftarrow b$ 
26:    Add  $\langle M[u], M[v] \rangle$  to  $C$ 
27:   $T \leftarrow \text{ConnectedComponents}(C)$ 
28:  for all  $v \in V$  in parallel do
29:     $M'[v] \leftarrow T[v]$ 

```

each bucket, we can sort by destination vertex using an arbitrary sequential sorting algorithm, processing each bucket in parallel. At the end of the sort phase, each vertex's operations are clustered within the batch into a group of deletions and a group of insertions pertaining to that vertex. At this stage, one could carefully reconcile matching insertions with deletions, which is especially important for multigraphs. For our experiments, we will skip reconciliation, processing each inserted and deleted edge, and allowing only the existence or non-existence of a single edge between each pair of vertices.

In Phase 2, the data structure update phase, the STINGER data structure is given the batch of insertions and deletions to be processed. For each vertex, deletions are handled first, followed by insertions. This ordering creates space in the data structure before insertions are added, minimizing the number of new blocks that must be allocated in the data structure and thereby reducing overhead.

After updating the graph, edge deletions identified earlier are checked to see if they disrupt connectivity in Phase 3. We create a bit array, in which each bit represents a vertex in the graph, for each unique source vertex in the batch of edge deletions. A bit set to 1 indicates that the vertex represented by that bit is a neighbor. Because of the scale-free nature of the graph, the number of bit arrays required for a batch is much less than the number of vertices. Since vertices can be involved in many edge deletions, the fine-grained synchronization available on the Cray XMT enables parallelism in the creation phase and re-use of the bit arrays in the query phase. We compute the intersection of neighbor sets by querying the neighbors of the sink vertices in the source bit array. Given that a social network is a scale-free graph, the rationale is that this intersection will quickly reveal that most of the edge deletions do not disrupt connectivity. Regarding running time and memory consumption, note that a common case bit array intersection for vertices with small degree can be handled by a quick lookup in the sorted list of neighbors and the bit matrix intrinsic of the

Cray XMT.

At this point, we can take the remaining edge deletion candidates and further process them to rule out or verify component structure change, likely using a breadth-first search. Otherwise, we will store the number of relevant deletions R seen thus far. After this number has reached a given threshold R_{thresh} determined by the tolerance for inconsistency before recomputation, we will re-compute the static connected components to determine the updated structure of the graph given the deletions that have taken place since the last static recomputation.

If we did not exceed R_{thresh} in the previous phase, the insertions must now be processed in Phase 4. For each edge being inserted, we look up the vertex endpoints in the current component mapping and replace them with the component ID to which they belong. In effect, we have taken the batch of edge insertions and converted it into a component graph. As this is a scale-free network, many of the insertions will now be self-edges in the component graph. The remaining edges will indicate components that have merged. Although it is unlikely, chains of edges, as well as duplicate edges, may exist in the batch. The order of merges is determined by running a static connected components computation on the new component graph¹. The result is an updated number of components in the graph and an updated vertex-component mapping.

In both the static connected components case and when finding the connected components of the component graph, we use an algorithm similar to Kahan’s algorithm [55]. Its first stage, performed from all vertices in the graph, greedily colors neighboring vertices using integers. The second stage repeatedly absorbs higher labeled colors into lower labeled neighbors. Colors are relabeled downward as another series of parallel breadth-first searches. When collisions between colors are no longer produced, the remaining colors specify the components.

¹In our implementation we use a compressed sparse row (CSR) representation, rather than creating a new graph data structure, as this only requires a sort of the batch of edges and a prefix sum, both done in parallel.

Table 6: Updates per second on a graph starting with 16 million vertices and approximately 135 million edges on 32 processors of a Cray XMT.

	Batch Size (edges)			
	10,000	100,000	250,000	1,000,000
Insertions Only	21,000	168,000	311,000	931,000
Insertions + STINGER	16,700	113,000	191,000	308,000
Insertions + STINGER + Bit Array	11,800	88,300	147,000	240,000
STINGER + Static Connected Components	1,070	10,200	22,400	78,600

Looking closely at the algorithm, one will note that when handling insertions only, the graph data structure does not need to be accessed. We can track the number of components and their sizes using only the vertex-component mapping. The insertions-only algorithm is very fast as the number of “vertices” in the component graph is small compared to the size of the original graph. Additionally, the number of “edges” in the component graph is bounded by the batch size. We observe insertions-only performance of up to 3 million updates per second on a graph with 1 million vertices and nearly 1 million updates per second on the larger graph with 16 million vertices – see Figure 6 for more data on the larger graph.

STINGER, the data structure, inevitably contributes a small overhead to the processing of updates, but it is scalable with larger batches. The update rate of insertions and the data structure can be viewed as an upper bound on the processing rate once deletions are introduced. One method that has been used for handling temporal updates is to re-compute static connected components after each edge or batch of edges. This update rate can be viewed as a lower bound on processing once deletions are introduced. Any method that will decrease the number or frequency of recomputations will increase the rate at which streaming edges can be processed.

We introduce the bit array intersection method as a means to rule out a large number of deleted edges from the list of deletions that could possibly affect the number of connected components. In the algorithm, we set a threshold R_{thresh} meaning that

we will tolerate up to R_{thresh} deletions in which the structure of the graph may have been affected before recomputing static connected components. The performance results for insertions plus STINGER plus the bit array intersection represent the update rate if $R_{\text{thresh}} = \infty$. Choosing R_{thresh} will determine performance between the lower bound and this rate. In practice, reasonable values of R_{thresh} will produce performance closer to the upper rate than the lower bound.

As an example of the effect of the bit array on the number of possibly relevant deletions, our synthetic input graph with 16 million vertices produces approximately 6,000 edge deletions per batch of 100,000 actions. The 12,000 endpoints of these deletions contain only about 7,000 unique vertices. Using a bit array to perform an intersection of neighbors, all but approximately 750 of these vertices are ruled out as having no effect on the graph. Performing a neighbor of neighbors intersection would likely reduce this number considerably again at the cost of increased complexity. As it is, the synthetic graph used for these experiments has an edge factor of 8 making it extremely sparse and a worst-case scenario for our algorithm. A real-world social network like Facebook would have an edge factor of greater than 100. In a scale-free network, this would reduce the diameter considerably making our bit array intersection more effective. In our graph, less than 1 percent of deletions cleave off vertices from the big component, while our bit array intersection algorithm rules out almost 90 percent of deletions at a small cost in performance. Given that we can tolerate R_{thresh} deletions between costly static recomputations, a reduction in the growth rate of R , the number of unsafe deletions, will increase the time between recomputations, increasing throughput accordingly.

The left plot in Figure 16 depicts update rates on a synthetic, scale-free graph with approximately 1 million vertices and 8 million edges. Looking at insertions only, or solely component merges without accessing the data structure, peak performance is in excess of 3 million updates per second. The STINGER implementation incurs a small

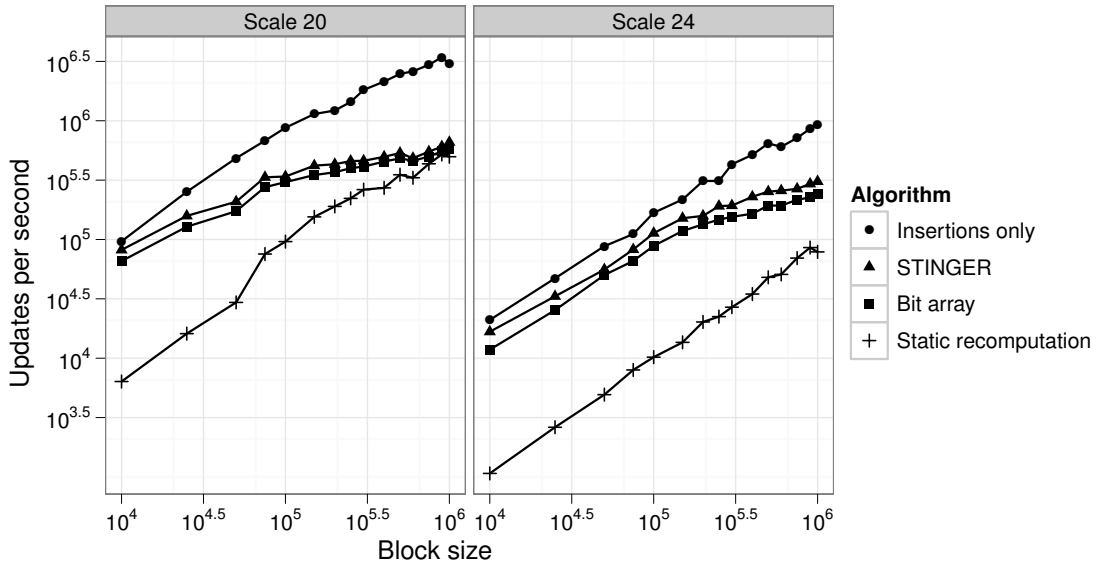


Figure 16: Update performance for a synthetic, scale-free graph with 1 million vertices (left) and 16 million vertices (right) and edge factor 8 on 32 processors of a 128 processor Cray XMT.

penalty with small batches that does not scale as well as the insertions-only algorithm. The bit array optimization calculation has a very small cost and its performance tracks that of the data structure. Here we see that the static recomputation, when considering very large batches, is almost as fast as the bit array. In this particular test case, we started with 8 million edges, and we process an additional 1 million edge insertions and/or deletions with each batch.

In Figure 16 on the right, we consider a similar synthetic, scale-free graph with approximately 16 million vertices and 135 million edges. The insertions only algorithm outpaces the data structure again, but by a smaller factor with the larger graph. The bit array performance again closely tracks that of the data structure. At this size, we can observe that the static recomputation method is no longer feasible, even for large batches. At this size, there is an order of magnitude difference in performance between the static connected components recomputation and the insertions-only algorithm.

We observe a decrease in performance in all four experiments as the size of the

graph increases from 1 million to 16 million vertices. There are several properties of the graph that change and affect performance. The larger graph has a larger diameter. Longer paths in the graph increase the running time of breadth-first search. As a result, the static connected components recomputation time increases. With more vertices to consider, the insertions only algorithm slows down when merging and relabeling components. The larger graph has a higher maximum degree, so walking the edge list of a vertex in STINGER will require additional time. Likewise, although the average degree remains fixed at 8, there are more vertices with degree larger than 8. Retrieving their neighbor list and performing the neighbor intersection will also see a performance penalty. Given that these steps are $O(n)$ in the worst case and are easily parallelized, we would expect good scalability with increasing graph size. In our experiments, the bit array optimization slowed by a factor of 2 when the graph grew by a factor of 16. The connected components recomputation slowed by a factor of 6 for the same increase in graph size.

CHAPTER V

ALTERNATIVE PROGRAMMING MODELS

5.1 MapReduce

The MapReduce [39] programming model first suggested by and demonstrated at Google is designed to ease parallel programming complexity. It is a data-parallel programming model in which the programmer provides sequential `map()` and `reduce()` functions that are applied to all data items. The `map()` and `reduce()` functions can be safely applied in parallel to distinct data blocks enabling scalability to the size of the data. MapReduce enables programmer productivity without heroic efforts on datasets whose volume may range in petabytes.

While it appears possible to express all algorithms, including graph algorithms, in this programming model, MapReduce exhibits several limitations that have practical effects on graph analysis codes. It is important to distinguish between the programming model and the implementation. While Hadoop [45] is a Java-based implementation of the MapReduce programming model, the discussion here will concentrate on characteristics of the programming model that are implementation-independent.

5.1.1 Data-parallelism and Locality

In the MapReduce programming model, data elements (or blocks) are stored across a distributed array of compute nodes. The job scheduler identifies the data blocks that will be required for a particular job and attempts to schedule the job tasks such that the computation is executed on the node that already contains the data. If this is not possible (or if a node “steals” work from another), the data block must be transferred over the network.

For a graph algorithm, the graph data is likely stored as a list of tuples or edges.

If the algorithm is edge parallel, the algorithm may map well to this representation. However, if the algorithm is vertex-centric, this representation will not take advantage of what little locality can be found in the algorithm.

The remedy to this problem is to use a preliminary MapReduce job to build a data representation analogous to compressed-sparse row in which the adjacency list for a particular vertex is stored as one data block. Subsequent MapReduce jobs will take these new data blocks as input at the cost of a redundant computation step and storage on disk.

Storing the graph natively in a compressed-sparse row-like representation limits the degree of parallelism that can be realized. Algorithms that exhibit behavior such as “for all vertices, for all neighbors” in parallel can take advantage of the Manhattan loop collapse, which MapReduce implementations do not yet exploit.

Using the adjacency list in this manner can also have significant impact on load balance in the system. A vertex with millions of adjacencies must be stored using a single data block on a single node. The MapReduce model will compute the `map()` task sequentially on the single node that contains the data block for that vertex, even if processing each neighbor could have been done in parallel on multiple nodes.

In this manner, the MapReduce programming model implicitly partitions graphs, either by edges or vertices. As previously discussed, small-world networks cannot be easily partitioned, and the programmer has no control over data placement.

5.1.2 Load Balance

MapReduce maps tasks to data and computes the task where the data lives. If the data is non-uniform in size, load imbalance can occur, even with highly dynamic work scheduling. In scale-free networks, the adjacency lists of vertices are non-uniform in length. If each task enumerates the neighbors of a vertex, the tasks will have uneven execution times and can suffer from load imbalance.

Load imbalance can also occur within the execution of an algorithm. In connected components, for example, each vertex or edge is computed on independently and mapped to a component label. Most vertices are in a single giant connected component. As component labels begin to converge, a larger percentage of the graph will exist in a single label and the tasks for that component label will be executed on a single machine.

When dynamic task scheduling and work stealing moves the computation from a heavily used machine to an idle one, the data that is necessary to do the computation must also move, unless the new machine contains a replica of the data block. Moving the computation information and also the data consumes time and network bandwidth resources.

5.1.3 Resilience

Other than ease of programming, a key contribution of the MapReduce model is its focus on resilience. Each `map()` and `reduce()` task is independent of all others and operates on independent data blocks. To accomplish resilience, data blocks are replicated across several nodes. If a node fails during the computation, the failure is detected by the job scheduler and the `map()` or `reduce()` task that was disrupted is restarted from another node containing a replica data block.

The obvious cost of resilience in this model is the replication of data blocks. If a system is to tolerate two simultaneous node failures, the data must be replicated by a factor of three, limiting the overall storage capacity or tripling the size of the system needed. There is an additional network cost for distributing replicas, but it is safe to assume that the rate of data ingest is lower than the rate of computation.

A more subtle cost of MapReduce's resilience to failure is the cost of intermediate data. As the input to each `map()` task is read from disk, so also is the output of the task written back to disk. The data is then sent to a `reduce()` task that writes its

output to disk. In the course of a single MapReduce job, only one additional copy of the intermediate data has been made for resilience purposes.

However most graph algorithms contain several MapReduce jobs. For breadth-first search, the number of jobs is proportionate to the diameter of the graph. In each case, the input to MapReduce job n is the output of job $n - 1$, which must be written to disk and then read back (we ignore the effect of the network and replica placement between jobs, although these are real factors). In an iterative algorithm, which are common in the graph world, the adjacencies must also be read from disk during each iteration.

If the quantity of intermediate data produced during an algorithm is much smaller than the input, then this model may be acceptable. However, we have observed graph algorithms that produce a combinatorial explosion of intermediate data. In the shared memory model, the intermediate data is often represented by doubly (or triply) nested loops and reads of the data. In the MapReduce model, the intermediate data must be explicitly stored and re-read.

5.1.4 Communication Costs

In the shared memory model, communication is implicit through read operations. Likewise, in the MapReduce programming model, communication is implicit. In this case, communication takes place between `map()` and `reduce()` tasks, and possibly also at the beginning and end of a MapReduce job.

The output of a `map()` task is a set of (key, value) pairs that must be aggregated by key. In most implementations, `map()` tasks and `reduce()` tasks occupy the same set of physical machines, although this is not required. In between phases, compute nodes perform a sort of their (key, value) pairs and then an all-to-all exchange so that a `reduce()` task receives all values of the key for which it must reduce.

The bandwidth required to make this exchange is potentially high as the worst case

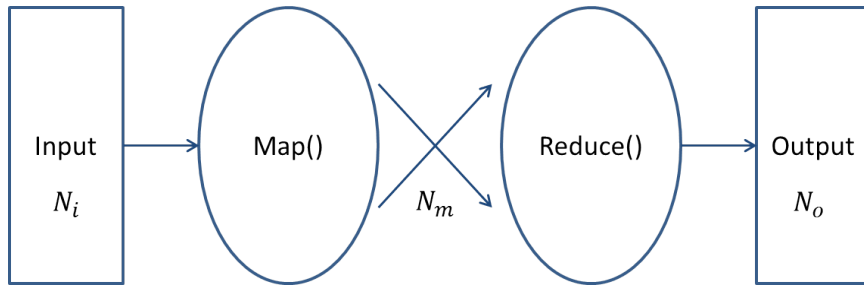


Figure 17: Workflow of a single MapReduce job.

involves every node sending all of its (key, value) pairs into the network and receiving all (key, value) pairs from other nodes. Load balance is of particular concern here as the number of (key, value) pairs to process is the result of the partition function between mappers and reducers. If the distribution is non-uniform, as may be the case in small-world graph analytics, the computation may wait on a single reducer.

The load balance issue could be resolved, or at the least greatly mitigated, by requiring `reduce()` tasks to be associative. In this case, reduction operations could be done locally at the node at which the `map()` task resided. Each node would perform a local reduction of its (key, value) pairs and then begin a reduction tree across all nodes in the computation. This would greatly reduce the communication bandwidth required, as well as take advantage of custom network hardware and software libraries for performing fast parallel reductions across machines. Newer versions of Hadoop implement “local reducers” to eliminate duplicate (key, value) pairs or perform the local part of the reduction before the shuffle phase when this is possible.

5.1.5 Complexity

There are five stages to a MapReduce job. These stages are illustrated in Figure 17.

The input stage reads data elements from files on disk. It reads N_i elements that are passed as input to the `map()` stage. The `map()` stage produces N_m outputs from the N_i inputs and sends these to the `reduce()` stage. In between these two stages, there is an intermediate phase sometimes referred to as the “shuffle” phase

in which results are sorted and distributed according to their keys. The `reduce()` phase takes N_m elements and produces N_o outputs that are written back to disk in the output stage.

The complexity of a MapReduce job depends on the number of data elements processed at each step. We assume the dominant cost of a MapReduce job is reading data elements from disk at the beginning and writing the result data to disk at the output. Network and communication costs are assumed to be small compared to data I/O. In an earlier section, we confronted a similar problem for shared memory graph analytics in GraphCT. To amortize the cost of loading data from disk, we performed as many operations as possible before terminating. Likewise, the goal of an ideal MapReduce job will be to compute as much as possible for the cost of a single I/O operation.

According to the defined goal, the ideal number of MapReduce jobs to compute a given analytic is one. If the answer can be found after reading the data a single time, it is efficient in terms of amortizing the dominant cost. Algorithms that complete in fewer iterations are considered better. Likewise, the number of output elements N_o ought to be less than or equal to the number of input elements N_i . A MapReduce job should seek to reduce the dimensionality of the input data. N_o should be less than or equal to N_m so that the I/O complexity of the output is the same or less than the input stage. In MapReduce, a “reduction” can be any operation on the (key, value) pairs of the `map()` stage. Often the `reduce()` function is a true reduction that summarizes all (key, value) pairs with a single value per key. In fact, requiring that the reduction be associative will eliminate the shuffle step prior to the `reduce()` and enable optimal reduction trees in parallel machines without moving (key, value) pairs.

Care should be taken that N_m not be much larger than N_i . In a graph algorithm, the input is often the adjacency lists of all vertices. Thus N_i is $|V|$. The `map()`

function then produces a (key, value) pair per neighbor, so N_m is $|E|$. Clearly $|E|$ is greater than $|V|$ by a factor equal to the average degree. However, it is possible to overwhelm a single node’s memory when $N_m \gg N_i$.

If we consider a MapReduce algorithm for connected components, N_m is always greater than N_i and N_o is equal to N_i , but the number of iterations is greater than one. For breadth-first search, N_m is usually greater than N_i , but N_o is often also greater than N_i . The number of iterations is bound by the diameter and greater than one. Both of these algorithms may not be good fits for MapReduce. Triangle enumeration completes in just one iteration and $N_o = N_i$. It is possible that N_m is extremely large, however. This algorithm is a better candidate for MapReduce.

5.1.6 Experimental Results

Real-world performance depends on many factors, including the size and complexity of the dataset, the number of machines and disks, the configuration of the machines, and the interconnection network characteristics. In this section, we will describe experimental results for breadth-first search, connected components, and clustering coefficients on Hadoop and other similar systems from the peer-reviewed literature.

Breadth-first search is among the most popular graph algorithms currently under study. Kang and Bader [46] built a hybrid workflow of a Hadoop-based cluster and a Sun UltraSparc T2 multithreaded system to study graph extraction and breadth-first search. The synthetic graph had 351 million vertices and 2.75 billion edges. Subgraph extraction on Hadoop took 23.9 hours and 30 breadth-first searches completed in 103 hours, or 3.4 hours per traversal. The Sun multithreaded system completed all 30 traversals in 2.62 seconds.

HADI [40] is a Hadoop-based tool for calculating effective diameter of a graph. On a synthetic graph with 2 billion edges, HADI runs in about 3.5 hours using 90 servers. Cluster servers are dual 4-core Intel Xeon with 8 GiB main memory and 3

TiB storage each.

Sector/Sphere [86] is a MapReduce-like project that contains a parallel file system and an on-disk computation engine. The authors constructed a dataset from PubMed articles that contained 28 million vertices and 542 million edges at 6 GiB. A larger dataset was constructed by replicating the existing dataset and adding random edges to build a graph with 5 billion vertices and 118 billion edges at 1.3 TiB. The test cluster has 60 dual 4-core Intel Xeon servers with 16 GiB main memory each and 4x1 TiB disk. Breadth-first search on the small graph took an average of 40 seconds. On the larger graph, the average time was 156 seconds. The authors report that the LexisNexis Data Analytics Supercomputer (DAS) took 120 seconds on the first query with 20 servers.

The Apache HAMA project [87] performs matrix operations and graph computations on top of Hadoop and other cloud services. Although execution time is only given for conjugate gradient, HAMA makes use of Hadoop MapReduce for dense matrix computations and their own proprietary bulk synchronous parallel (BSP) engine or Microsoft's Dryad [88] engine for graph computations, citing more flexible communication patterns.

Looking for a theoretical framework from which to analyze MapReduce algorithms, Pace [89] proves that MapReduce algorithms can be implemented under the bulk synchronous parallel (BSP) model of computation and analyzed as such. Using this theoretical framework, he shows that breadth-first search cannot efficiently be implemented in MapReduce as the entire graph must be stored and read in each superstep. Load imbalance in scale-free networks will overwhelm a single processor. In the BSP model of computation, the level synchronous breadth-first search is immune to this issue.

Several algorithms have been put forward for connected components in MapReduce. CC-MR [90] compares against Pegasus [63] and Cohen's algorithm [41] on a 14

server cluster with each node having 8 cores and 16 GiB of main memory. CC-MR performs at least a factor of two faster than Pegasus or Cohen. The IMDb graph with 20 million edges runs in 1055 seconds on CC-MR, 3033 seconds for Cohen, and 2834 seconds under Pegasus.

Calculating clustering coefficients, or counting triangles, requires many neighborhoods to be intersected. As opposed to connected components and breadth-first search, which are iterative, triangle counting algorithms in MapReduce typically complete in a single iteration, but can produce large quantities of intermediate data. Wu finds the clustering coefficient of a phone call network with Hadoop on 32 4-core compute nodes with 8 GiB main memory each in 160 seconds. The graph has 1.6 million vertices and 5.7 million edges [91].

Suri [92] offers algorithms for clustering coefficients using larger graphs on a larger cluster. Using a Hadoop cluster with 1636 servers, the clustering coefficients of the LiveJournal dataset with 4.8 million vertices and 6.9 million edges are found in 5.33 minutes. The larger Twitter dataset has 42 million vertices and 1.5 billion edges and computes in 423 minutes.

In our shared memory work with GraphCT and STINGER on the Cray XMT and Intel platforms, we have demonstrated how to compute the same metrics on comparable or larger graphs in much less time. We can compute a breadth-first search in 0.3 seconds on a graph that has 16 million vertices and 268 million edges (a similar graph in [86] took 156 seconds). Connected components on a graph with 2.1 billion vertices and 17.1 billion edges takes 86.6 seconds (12x faster than [90] on data 855x larger). The Cray XMT finds all triangles in a scale-free network with 135 million edges in 160 seconds ([92] takes 160x longer on a graph 11x larger).

It is now clear that graph computations in MapReduce frameworks, like Hadoop and Sphere, are orders of magnitude slower than in shared memory applications, although it is still uncertain if this results from the hardware, software, or algorithm

itself. The level synchronous breadth-first search is provably inefficient under MapReduce. In the next section, we will examine the more flexible bulk synchronous parallel (BSP) programming model and the algorithmic effects of BSP on graph applications before returning to the hardware question.

5.2 *Bulk Synchronous Parallel and Pregel*

Despite its extensive use of MapReduce, Google has put forward an alternative framework for graphs called Pregel [58]. Pregel is based on the bulk synchronous parallel (BSP) style of programming. It retains many of the properties of MapReduce, including simple sequential programming and resilience, but adds active messaging and a vertex-centric view of the graph with maintained state between supersteps.

Bulk synchronous parallel programming is popular in the scientific community and is the basis for many large-scale parallel applications that run on today's supercomputers. An application is composed of a number of *supersteps* that are run iteratively. Each superstep consists of three phases. In the first phase, nodes process incoming messages from the previous superstep. In the second phase, nodes compute local values. In the third phase, nodes send messages to other nodes that will be received in the next superstep. The restriction that messages must cross superstep boundaries ensures that the implementation will be deadlock-free.

Applying bulk synchronous parallel programming to graph analytics is straightforward. Each vertex becomes a first-class citizen and an independent actor. The vertex implicitly knows its neighbors (they do not have to be read from disk each iteration). Each vertex is permitted to maintain state between supersteps. A vertex can send a message to one or all of its neighbors or to any other vertex that it can identify (such as through a message that it has received). If a vertex has no messages to send or nothing to compute, it can vote to stop the computation, and will not be re-activated until a future superstep in which it has messages to receive.

The introduction of vertex state, as well as the representation of the graph edges in the communication patterns, removes two of the greatest difficulties for expressing graph algorithms in MapReduce. We will now present easy transformations for classical shared memory graph algorithms into the BSP model.

5.2.1 Experimental Method

The primary focus of this experiment is to determine the algorithmic effects of the bulk synchronous parallel programming model on the graph algorithms described above. Google’s Pregel platform is not publicly available. Open source alternatives exist that run on top of Hadoop, such as Apache Giraph [93], but too many differences exist between the Hadoop implementation and fundamental C-language shared memory programming.

To remove as many differences as possible between the BSP implementation and the shared memory implementation of these graph algorithms, we devised a bare-bones, shared memory BSP environment written in C, and built on top of GraphCT [54]. The BSP environment supports messages between vertices, as well as messaging to all neighbors. Incoming and outgoing message queues are transferred between iterations with a single pointer swap. An array of vertex states is maintained between iterations as well as an array of vertex votes. With each superstep, the BSP function is called to execute the superstep. To measure performance, we track the time to execute each superstep as well as the number of messages sent and received.

The comparison code is a hand-written C-language implementation of the graph algorithms. The implementations are instrumented to measure the time per iteration, time to solution, as well as the number of reads and writes. Both the shared memory and BSP implementations are run on the same input graph on the same machine and we examine intermediate data for correctness.

Algorithm 4 A parallel algorithm for connected components in the BSP model.

Input: Superstep s , Vertex v , Current label L , Set of Messages M_v

Output: Outgoing message set M'_v

```
1:  $Vote \leftarrow 0$ 
2: for all  $m \in M_v$  do
3:   if  $m < L$  then
4:      $L \leftarrow m$ 
5:    $Vote \leftarrow 1$ 
6: if  $s = 0$  then
7:    $L \leftarrow \min(M_v)$ 
8:   for all  $n \in Neighbors(v)$  do
9:     Send  $L$  to  $n$ 
10: else
11:   if  $Vote = 1$  then
12:     for all  $n \in Neighbors(v)$  do
13:       Send  $L$  to  $n$ 
```

5.2.2 Connected Components

Connected components is a reachability algorithm that labels all vertices in a connected component with the same level. An algorithm for connected components in the BSP model is shown in Algorithm 4. Each active vertex will execute this algorithm for each superstep.

For this algorithm, the vertex state will store the component label of the component to which each vertex belongs. In the first superstep, each vertex will set its state (label) to be itself, or each vertex will begin by belonging to its own component, as in the Shiloach-Vishkin approach [42]. Each vertex then sends its component label to all neighbors.

In each subsequent superstep, all active vertices will receive their incoming messages and check each one to see if it contains a component label that is smaller than the current state. If it finds such a new component label, it will update the current state and send the new component label to all of its neighbors, to be received in the next superstep. When all vertices have found no changes and have voted to stop the

computation, the algorithm terminates with the correct vertex-component mapping.

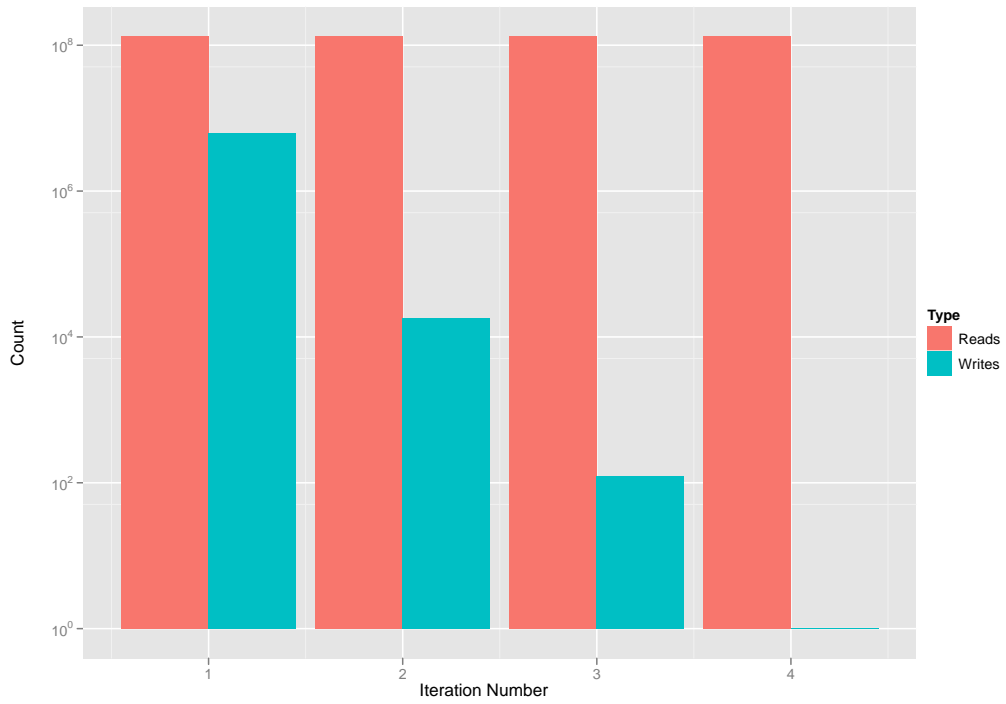
We run connected components on an undirected, scale-free RMAT [73] graph with 2 million vertices and 131 million edges. The shared memory implementation from GraphCT completes in four iterations. In Figure 18(a), each iteration performs 132 million reads. A write is performed when a component label change is detected. The number of writes recorded drops quickly from over 6 million in the first iteration to only 121 in the third iteration. The total ratio of reads to writes is approximately 85:1.

In Figure 18(b), the number of iterations that the BSP components algorithm takes is 10. In the BSP algorithm, the only vertices that are active are those that received messages from the previous superstep. In the first several supersteps, nearly all vertices are active and changing their component labels. Beginning with the fifth superstep, the number of component label changes drops off dramatically. However, the ratio of total reads to total writes for BSP is 1.3:1.

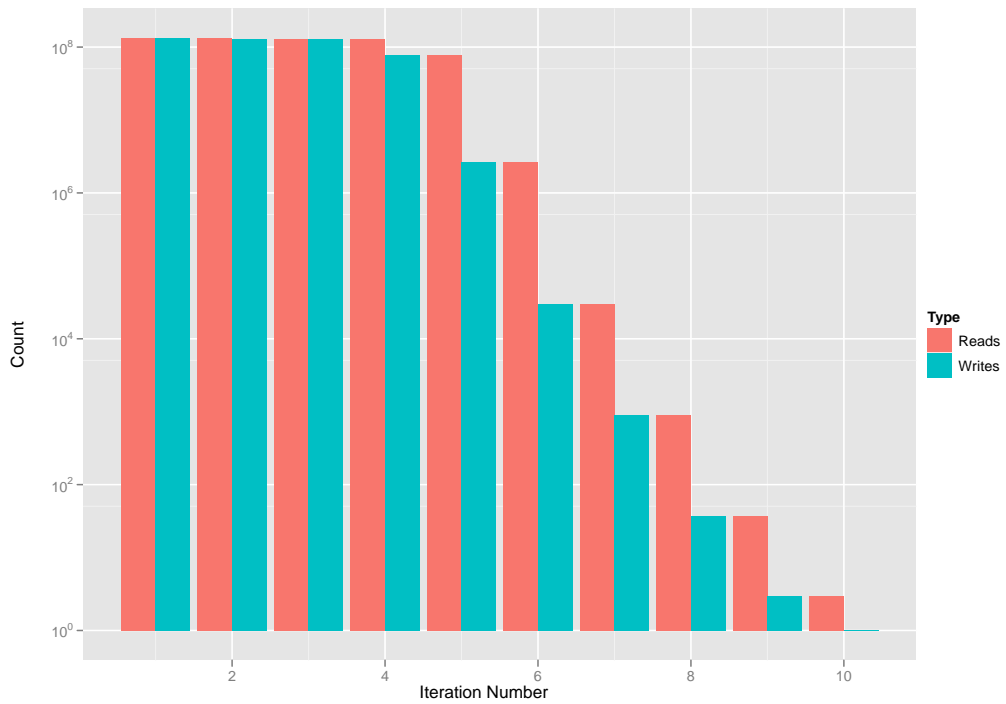
Using a single core on a dual socket 2.4 GHz Intel Xeon E5530 with 12 GiB of main memory, the shared memory connected components takes 6.83 seconds compared to the BSP implementation that completes in 12.7 seconds. The shared memory implementation processes at a rate of 19.3 million edges per second compared to 10.4 million edges per second for the BSP.

Despite the fact that the BSP connected components algorithm computes only on the active vertex set and performs only 15 percent more reads than the shared memory connected components, it requires double the number of iterations and takes twice as long to complete.

Note that the time per iteration for the shared memory implementation is constant in Figure 19. In the BSP model, the time is proportional to the number of messages being sent and received. Early iterations take two orders of magnitude longer than later iterations. However the early iterations are too long to make this approach



(a) Shared Memory Connected Components



(b) Bulk Synchronous Parallel Connected Components

Figure 18: The number of reads and writes per iteration performed by connected components algorithms on a scale-free graph with 2 million vertices and 131 million edges.

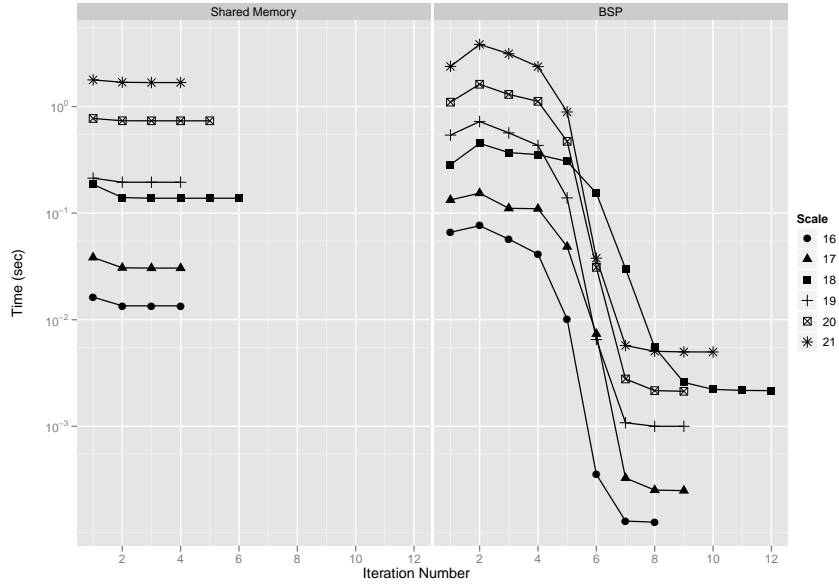


Figure 19: Execution time per iteration performed by connected components algorithms. Scale is the log base 2 of the number of vertices and the edge factor is 8. For Scale 21, shared memory completes in 8.3 seconds, while BSP takes 12.7 seconds.

competitive with the shared memory implementation.

The parallel execution time on the Cray XMT for each iteration of connected components is plotted in Figure 20. The input graph is an undirected, scale-free RMat [73] graph with 16 million vertices and 268 million edges. On the left, the BSP algorithm completes in 13 iterations. In the first four iterations, almost all vertices are active, sending and receiving label updates. As the labels begin to converge, the number of active vertices, and execution time, drops significantly. In the last six iterations, only a small fraction of the graph is active.

On the right, the shared memory algorithm in GraphCT completes in 6 iterations. The amount of work per iteration is constant, and the execution time reflects this. Label propagation early in the algorithm reduces the number of iterations compared to the BSP algorithm, which uses four iterations to resolve the majority of the graph. In the shared memory algorithm, most labels are fixed by the end of the first iteration.

Each line in Figure 20 corresponds to performance for a different Cray XMT

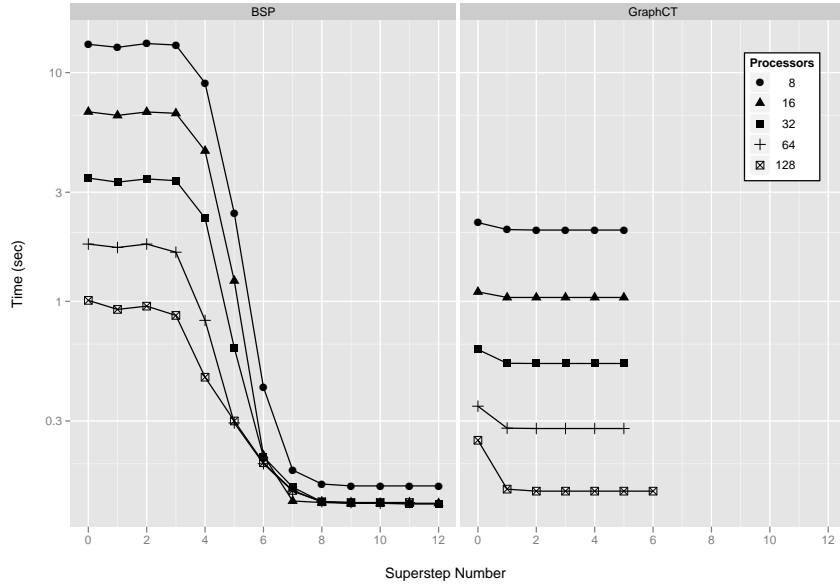


Figure 20: Connected components execution time by iteration for an undirected, scale-free graph with 16 million vertices and 268 million edges. On the 128-processor Cray XMT, BSP execution time is 5.40 seconds and GraphCT execution time is 1.31 seconds.

processor count. Time is on a log scale and the number of processors doubles with each line. Even vertical spacing between points indicates linear scaling for a given iteration. All iterations of the shared memory algorithm in GraphCT demonstrate linear scaling. In the BSP algorithm, the first iterations that involve the entire vertex set also have linear scaling. As the number of active vertices becomes small, the parallelism that can be exposed also becomes small and scalability reduces significantly.

The total time to compute connected components on a 128-processor Cray XMT using GraphCT is 1.31 seconds. The time to compute connected components using the BSP algorithm on the Cray XMT is 5.40 seconds.

5.2.3 Breadth-first Search

Breadth-first search, the classical graph traversal algorithm, is more recently used in the Graph500 benchmark [94]. In the BSP algorithm, the vertex state stores the current distance from the source. In the first superstep, the source vertex sets it

distance to zero, and then sends its distance to all of its neighbors. All other vertices begin with a distance of infinity. Algorithm 5 gives a description in the BSP model. Each active vertex will execute this algorithm for each superstep.

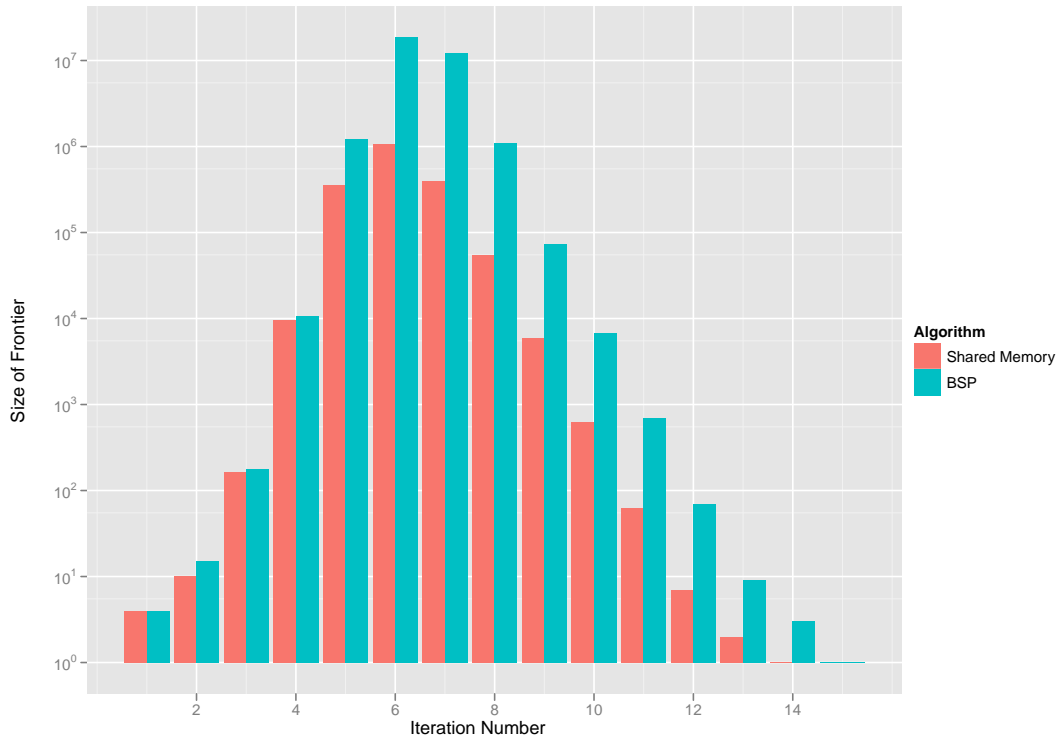
In each subsequent superstep, all vertices that are potentially on the frontier will receive messages from the previous superstep and will become active. If the current distance is infinity, the vertex will process the incoming messages and set the distance appropriately. The vertex then sends its new distance to all of its neighbors, and votes to stop the computation.

The computation continues to iterate as long as vertices have a distance that is infinity. Once all vertices have computed their distance, the computation can terminate.

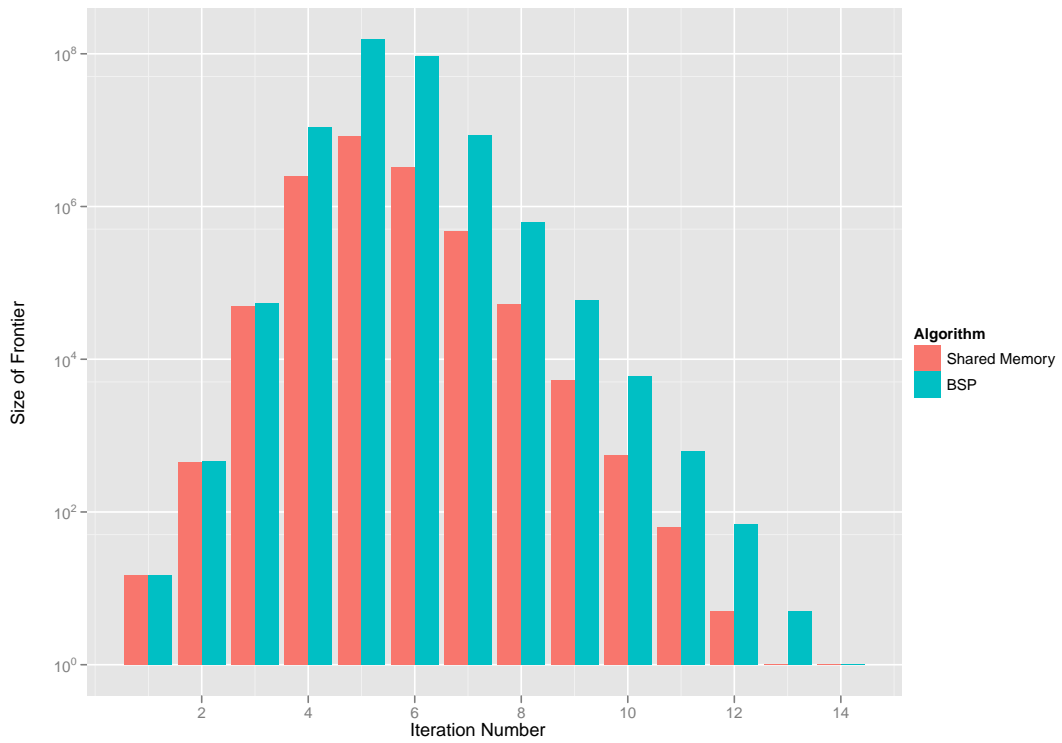
The BSP algorithm closely mimics the level-synchronous, parallel shared memory breadth-first search algorithm [1] with the exception that messages are sent to vertices that *may be* on the frontier, while the shared memory algorithm enqueues only those vertices that are definitively unmarked and on the frontier. The level-synchronous shared memory breadth-first search algorithm is the comparison in GraphCT in Figures 21 and 22.

We run breadth-first search on an undirected, scale-free RMat [73] graph with 2 million vertices and 33 million edges. The shared memory implementation from GraphCT completes in 14 iterations. In Figure 21, the number of vertices on the frontier is shown as a function of the iteration number. The frontier grows exponentially and peaks in the sixth iteration. Afterward, it shrinks exponentially from 1 million vertices to just six vertices in the twelfth iteration.

In contrast, the number of messages generated by each BSP superstep is plotted in Figure 21. In the BSP algorithm, a message is generated for every neighbor of a vertex on the frontier. Initially, almost every neighbor of the frontier is on the next frontier, and the number of messages is approximately equal to the size of the frontier.



(a) Scale-free graph with 2 million vertices and 33 million edges.



(b) Scale-free graph with 16 million vertices and 268 million edges.

Figure 21: Size of the frontier as a function of breadth-first search level.

Algorithm 5 A parallel algorithm for breadth-first search in the BSP model.

Input: Superstep s , Vertex v , Current distance D , Set of Messages M_v

Output: Outgoing message set M'_v

```
1:  $Vote \leftarrow 0$ 
2: for all  $m \in M_v$  do
3:   if  $m + 1 < D$  then
4:      $D \leftarrow m + 1$ 
5:      $Vote \leftarrow 1$ 
6: if  $s = 0$  then
7:   if  $D = 0$  then
8:      $Vote \leftarrow 1$ 
9:   for all  $n \in Neighbors(v)$  do
10:    Send  $D$  to  $n$ 
11: else
12:   if  $Vote = 1$  then
13:     for all  $n \in Neighbors(v)$  do
14:      Send  $L$  to  $n$ 
```

As the majority of the graph is found, messages are generated to vertices that have already been touched. The number of messages from superstep four to the end is an order of magnitude larger than the real frontier. However, the number of messages does decline exponentially.

For a given superstep, the difference between the number of messages in the BSP model and the size of the frontier in the shared memory model is equal to the number of vertices that are checked but are already marked in the shared memory algorithm. In shared memory, these are reads whereas in BSP they are writes followed by reads. As the average density of the graph grows, we would expect the gap to grow.

Using a single core of a 3.4 GHz Intel Core i7-2600K with 16 GiB of main memory, the shared memory breadth-first search takes 772 milliseconds compared to the BSP implementation that completes in 803 milliseconds. The graph is scale-free with 2 million vertices and 33 million edges. Despite the fact that the number of messages generated in BSP is an order of magnitude larger than the shared memory's writes to the frontier, the execution time is comparable.

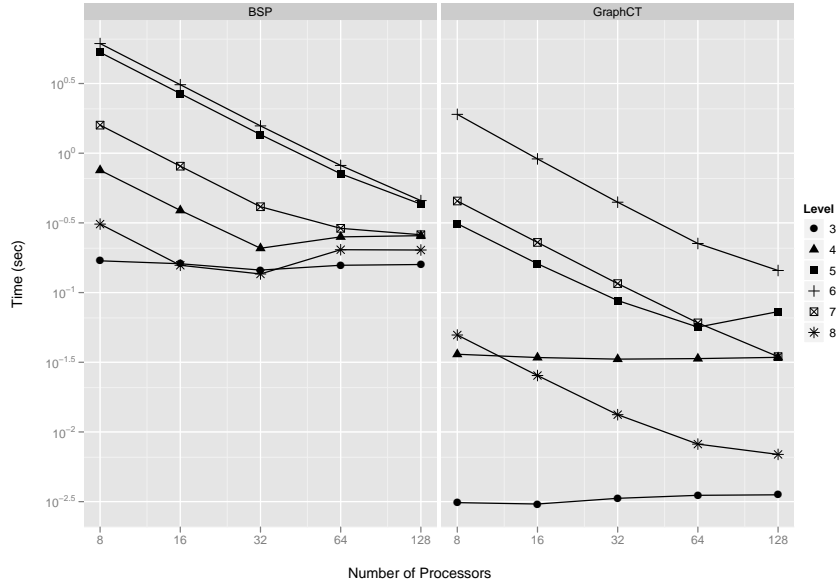


Figure 22: Scalability of breadth-first search levels 3 to 8 on an undirected, scale-free graph with 16 million vertices and 268 million edges. Total execution time on the 128-processor Cray XMT is 3.12 seconds for BSP and 310 milliseconds for GraphCT.

Given that the graph is in compressed-sparse row format, the frontier and the message queue are contiguous, the sequential algorithm has good spatial locality in the data structure. The largest frontier is 8.5 MiB and the largest message list is 30 MiB. The processor L3 cache is 8 MiB. Given the exponential rise and fall of both, the working set for most iterations fits primarily in cache.

We consider parallel scaling of the breadth-first search algorithms on the Cray XMT in Figure 22. The input graph is an undirected, scale-free RMAT [73] graph with 16 million vertices and 268 million edges.

Scalability is directly related to parallelism, which is itself related to the size of the frontier. The scalability of iterations 3 through 8 is plotted in Figure 22. The early and late iterations show flat scaling and are omitted. On the right, the GraphCT breadth-first search has flat scalability in levels 3 and 4. Levels 5 and 8 have good scalability to 64 processors, but reduce at 128, implying contention among threads. Levels 6 and 7 have linear scalability, which matches the apex of the frontier curve.

Likewise, the BSP scalability on the left of the figure shows flat scaling in levels 3 and 8. Levels 4 and 8 have good scalability at 32 processors, but tail off. Levels 5, 6 and 7 have almost linear scalability. Because the number of messages generated is an order of magnitude larger than the size of the frontier, the contention on the message queue is higher than in the GraphCT breadth-first search and scalability is reduced as a result. Overall, the innermost BSP levels have similar execution times and scalability as the shared memory algorithm. However, the overhead of the early and late iterations is two orders of magnitude larger.

The total time to compute a single breadth-first search on a 128-processor Cray XMT using GraphCT is 310 milliseconds. The time to compute a breadth-first search from the same vertex using the BSP algorithm on the Cray XMT is 3.12 seconds. Typically, multiple breadth-first searches are executed in parallel on the Cray XMT to increase the available parallelism and reduce contention on a single frontier queue of vertices.

5.2.4 Clustering Coefficients

The computationally challenging aspect of computing the clustering coefficients of a graph is counting the number of triangles. In the shared memory model, this is the intersection of the neighbor list of each vertex with the neighbor list of each of its neighbors, for all vertices in the graph. In MapReduce, one approach is for each vertex to produce all possible pairs of its neighbors, or to enumerate the edges that would complete the triangles. A subsequent task then checks to determine the existence or non-existence of each of these edges.

The BSP algorithm takes a third approach. First, a total ordering on the vertices is established such that $V = v_1, v_2, v_3, \dots, v_N$. We define a triangle as a triple of vertices $\langle v_i, v_j, v_k \rangle$ such that $i < j < k$. We will count each triangle exactly once. The algorithm details are given in Algorithm 6. Each active vertex will execute this

Algorithm 6 A parallel algorithm for triangle counting in the BSP model.

Input: Superstep s , Vertex v , Set of Messages M_v **Output:** Outgoing message set M'_v

```
1: if  $s = 0$  then
2:   for all  $n \in \text{Neighbors}(v)$  do
3:     if  $v < n$  then
4:       Send  $v$  to  $n$ 
5: if  $s = 1$  then
6:   for all  $m \in M_v$  do
7:     for all  $n \in \text{Neighbors}(v)$  do
8:       if  $m < v < n$  then
9:         Send  $m$  to  $n$ 
10: if  $s = 2$  then
11:   for all  $m \in M_v$  do
12:     if  $m \in \text{Neighbors}(v)$  then
13:       Send  $m$  to  $m$ 
```

algorithm for each superstep.

In the first superstep, all vertices send a message to all neighbors whose vertex ID is greater than theirs. In the second superstep, for each message received, the message is re-transmitted to all neighbors whose vertex ID is greater the vertex that received the message. In the final step, if a vertex receives a message that originated at one of its neighbors, a triangle has been found. A message can be sent to itself or to another vertex to indicate that a triangle has been found.

Although this algorithm is easy to express in the model, the number of messages generated is much larger than the number of edges in the graph. This has practical implications for implementing such an algorithm on a real machine architecture.

We calculate the clustering coefficients of an undirected, scale-free RMAT [73] graph with 16 million vertices and 268 million edges. This algorithm is not iterative. It is a triply-nested loop in the shared memory implementation. The outer loop iterates over all vertices. The middle loop iterates over all neighbors of a vertex. The inner-most loop iterates over all neighbors of the neighbors of a vertex.

The BSP algorithm replaces the triply-nested loop with three supersteps. The first two supersteps enumerate all possible triangles (restricted by a total ordering). The third and final superstep completes the neighborhood intersection and enumerates only the actual triangles that are found in the graph.

Both algorithms perform the same number of reads to the graph. The BSP algorithm must emit all the possible triangles as messages in the second superstep. For the graph under consideration, this results in almost 5.5 billion messages generated. In the last superstep, we find that these 5.5 billion possible triangles give only 30.9 million actual triangles. It is worth noting that the RMAT graph under consideration contains far fewer triangles than a real-world graph. The number of intermediate messages will grow quickly with a higher triangle density.

The shared memory implementation, on the other hand, only records a write when a triangle is detected. The total number of writes is 30.9 million, compared with 5.6 billion for the BSP implementation. The BSP clustering coefficient implementation produces 181 times as many writes as the shared memory implementation.

The execution time and scalability of the GraphCT and BSP triangle counting algorithms on a 128-processor Cray XMT is plotted in Figure 23. The BSP implementation scales linearly and completes in 444 seconds on 128 processors. The shared memory implementation completes in 47.4 seconds on 128 processors.

5.2.5 Discussion

By implementing BSP in a C-language environment on the same shared memory platform on which we conduct our GraphCT experiments, we can observe the algorithmic differences imposed by the bulk synchronous parallel programming model. The total execution times for each algorithm on the Cray XMT is compared in Table 7. The Cray XMT enables scalable, parallel implementations of graph algorithms in both programming models. In some cases, such as connected components, the scalability

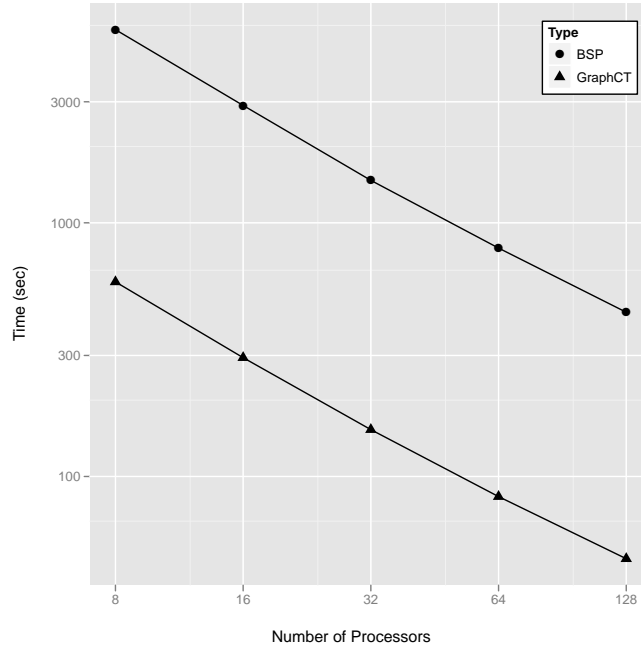


Figure 23: Scalability of triangle counting algorithms on an undirected, scale-free graph with 16 million vertices and 268 million edges. Execution time on the 128-processor Cray XMT is 444 seconds for BSP and 47.4 seconds for GraphCT.

and execution profiles are quite different for the same algorithm. In others, such as breadth-first search and triangle counting, the main execution differences are in the overheads, both memory and time.

Breadth-first search is the BSP algorithm that bears the most resemblance to its shared memory counterpart. Both operate in an iterative, synchronous fashion. The only real difference lies in how the frontier is expressed. The shared memory algorithm only places vertices on the frontier if they are undiscovered, and only places one copy

Table 7: Execution times on a 128-processor Cray XMT for an undirected, scale-free graph with 16 million vertices and 268 million edges.

Algorithm	Time (sec.)		Ratio
	BSP	GraphCT	
Connected Components	5.40	1.31	4.1:1
Breadth-first Search	3.12	0.31	10.0:1
Triangle Counting	444	47.4	9.4:1

of each vertex. The BSP algorithm does not have this knowledge, so it must send messages to every vertex that could possibly be on the frontier. Those that are not will discard the messages.

As a result, the two algorithms perform very similarly. In fact, many of the fastest performing Graph500 [94] entries on large, distributed clusters perform the breadth-first search in a bulk synchronous fashion with varying 1-D and 2-D decompositions [66]. While at a small scale on a cache-based system, the disparity in frontier size between the two algorithms does not seem to have a large impact on execution time, on a large graph in a large system, the network will be quickly overwhelmed by the order of magnitude greater message volume.

In the connected components algorithms, we observe different behavior. Since messages in the BSP model cannot arrive until the next superstep, vertices processing in the current superstep are processing on stale data. Because data cannot be forwarded in the computation, the number of iterations required until convergence is at least a factor of two larger than in the shared memory model. In the shared memory algorithm, once a vertex discovers its label has changed, that new information is available to all of its neighbors immediately and can be further consumed. While the shared memory algorithm requires edges and vertices to read and processed that will not change, the significantly lower number of iterations results in a significantly shorter execution time.

In the clustering coefficients algorithms, we observe very similar behavior in reading the graph. Each vertex is considered independently and a doubly-nested loop of the neighbor set is required (although the exact mechanisms of performing the neighbor intersection can be varied—see [28]). The most significant difference between the algorithms is the nature of the possible triangles. In the shared memory algorithm, the possible triangles are implicit in the loop body. In the BSP algorithm, the possible triangles must be explicitly enumerated as messages. The result is an overwhelming

number of writes that must take place. Despite 181 times greater number of writes, the Cray XMT only experiences a 9.4x slow down in execution time.

The bulk synchronous parallel model of graph computation is clearly easy to program and easy to reason about. However, the cost is a loss of flexibility in data movement and communication. Communication is now explicit, has a non-trivial memory footprint, and limits the rate at which information can flow through the algorithm. By contrast, shared memory’s communication is implicit, but the number of reads and writes is no greater than the explicit communication in BSP. In the next section, motivated by our knowledge of shared memory and BSP graph algorithms and their data access behavior, we will analyze the effect of hardware device characteristics on application performance.

5.3 Random Access and Storage Performance

Two central features of Hadoop (and the MapReduce model) are resilience to failure and scalable storage. To accommodate larger data sizes and computations, one only needs to add machines (and disks) to the cloud. As the number of disks increases, failure of a single component is more likely. Data replication partially mitigates this issue. Hadoop also reads from and stores all values to disk between phases of the computation so that lost tasks can be replayed. Graph algorithms involve a combination of random access and linear access patterns to memory. In this section, we will look at each storage technology, the access pattern of several algorithms, and the effect on computation time.

5.3.1 Measuring Data Access Time

Current high performance graph applications perform well in main memory. However, the access pattern in memory can have a dramatic effect on execution time. To measure the random access time to main memory, we utilize a simple microbenchmark. The microbenchmark takes an array of 64-bit integers of length N in memory and

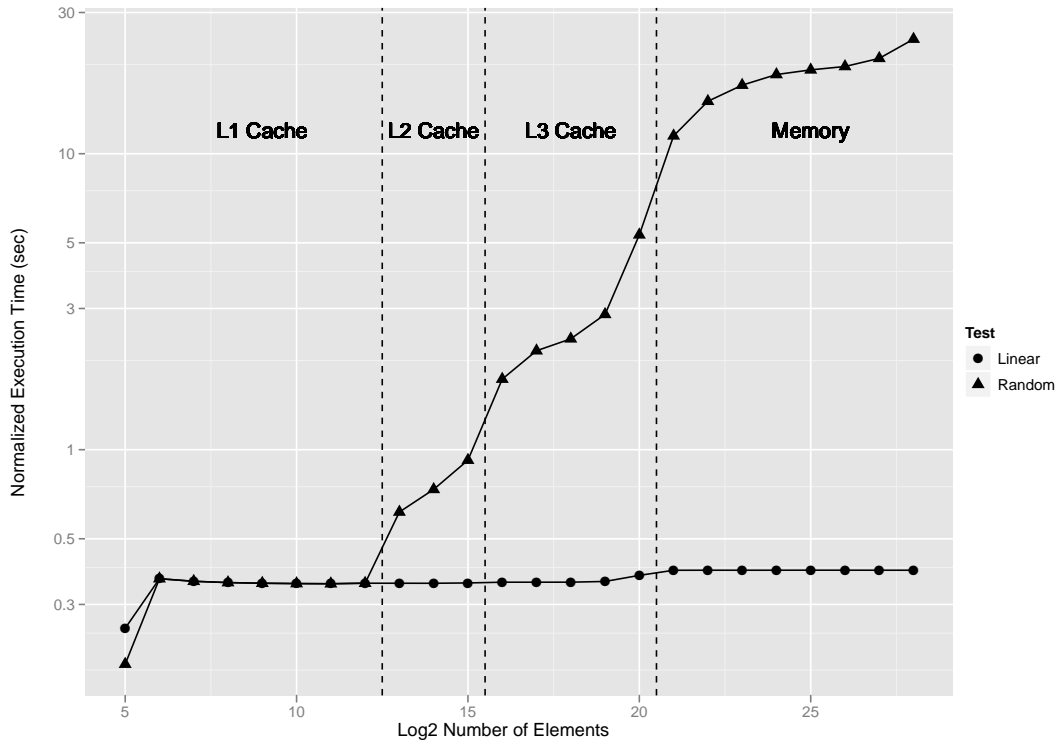


Figure 24: Normalized execution time for an array memory access microbenchmark.

traverses the array in a circular fashion, reading 2^{28} elements. We report the time to read all elements. We traverse the array in two different manners: linear (or stride-1) and random. In the linear traversal, we first read element zero, then one, and so on. In the random traversal, we read each element in a uniform random order.

Measurements are taken on a 3.4 GHz Intel Core i7-2600K with 16 GiB of main memory. In Figure 24, both linear and random microbenchmark trials run in about 350 milliseconds when the size of the array is 4096 elements or less. From this point, the time required for the random traversal increases significantly, while the time required for the linear traversal remains constant. At 268 million elements, the sequential access takes 391 milliseconds while the random access takes 24.4 seconds, or 62.4x longer.

In the random trial, we observe gaps in the data points that are larger than others (namely between 4096 and 8192, 32768 and 65536, 1M and 2M). Consulting

the Intel datasheet for this processor, these gaps correspond to the the transition between caches levels. In the linear case, the pre-fetching mechanisms of the cache hierarchy accurately predict the access pattern and the data is made available close to the processor.

From the data collected, we estimate that the time required for access to a random data element is 1.31 nanoseconds in L1 cache, 2.73 nanoseconds in L2 cache, 10.7 nanoseconds in L3 cache, and 69.0 nanoseconds in main memory. An algorithm with an extreme amount of locality will have a performance scalability in data footprint more closely resembling that of the linear traversal, while an algorithm with less locality will tend to require more time per element as the data size grows.

Applying the same experimental protocol to graph algorithms will further shed light on the degree of their locality. In this experiment, we time only the main computation (ignoring pre- and post-processing) for a variety of input graph sizes as measured by the number of edges. The input graphs are synthetic RMAT graphs with power-law distributions. We collect both the execution time and the number of edges read, and then normalize by the same 2^{28} factor as in the microbenchmark.

The results of the graph algorithm performance are depicted in Figure 25. The x-coordinate for the algorithmic data points is the log of the size of the memory footprint (as measured by the data fields that the algorithm actually reads). Unusually high data points in the first several graphs are likely due to program execution noise that is amplified by the normalization process.

Both breadth-first search and connected components exhibit behavior in which the normalized time to completion rises as a function of graph size. Both outperform the random traversal microbenchmark, but move in the same direction. The cache boundaries are less noticeable in these algorithms. In the largest graph considered, breadth-first search is 2.7 times faster than the random traversal. Connected components is 8.3 times faster. From this data, we conclude that breadth-first search

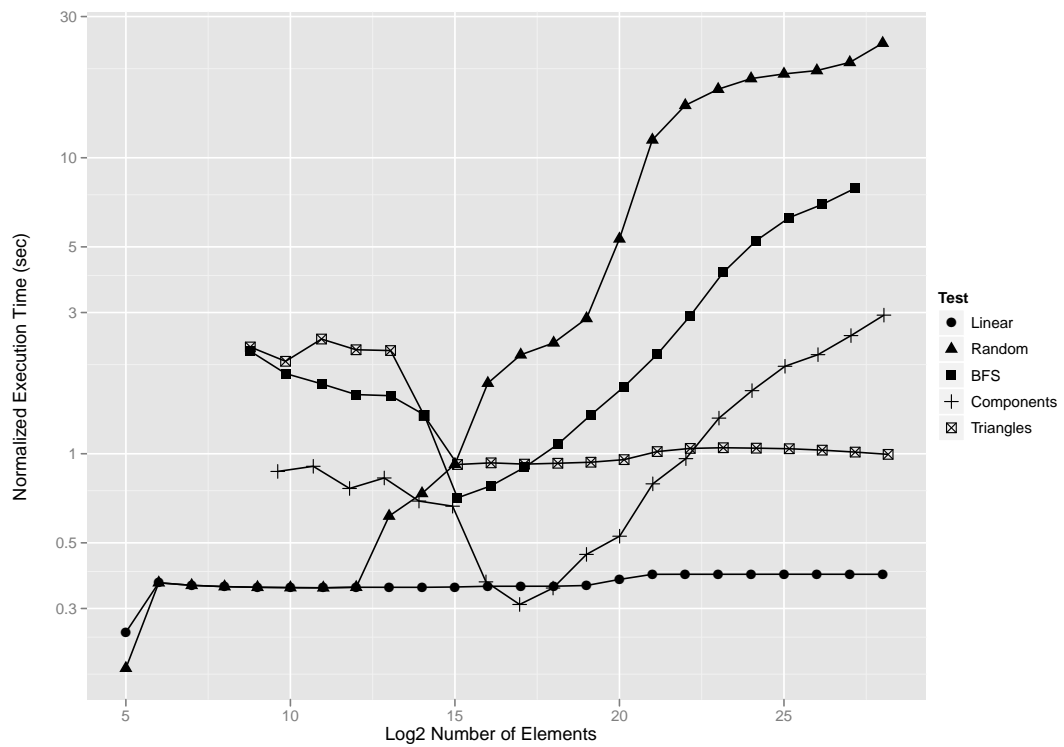


Figure 25: Normalized execution time comparing several static graph algorithms with linear and random array memory access.

and connected components both have more locality than the random traversal, but in general have a substantially random access pattern.

Clustering coefficients, or triangle counting, exhibits different properties. The normalized performance remains constant regardless of graph size. At the largest graph, it is 2.5 times slower than the linear microbenchmark. This seems to indicate a great deal of locality in the access pattern. In fact, the code itself consists of a large number of sorted list comparisons that highly resemble stride-1 access. Hidden in this view of the data is the number of edges that must be considered. The largest graph, with 267 million edges, requires 65.6 billion edges read, or 245 complete passes over the graph. The number of passes required over the edge set appears to grow exponentially with graph size, although with a small exponent.

We can verify these experimental findings by looking at the algorithms directly. For connected components, in each iteration we read $|E|$ source vertices and $|E|$ destination vertices, which are both stored and accessed contiguously. We also read $|V|$ source component labels and $|V|$ destination component labels, which are random accesses. For a graph with an average degree of 8, we estimate that 11 percent of reads will be random and 89 percent of reads will be linear.

Likewise for breadth-first search, reading the graph neighbors will require $|V|$ random reads (the offset into the edge list) and $|E|$ linear reads (the edges themselves). Checking the distance of each neighbor is $|V|$ random reads. Adding a vertex to the frontier is $|V|$ random writes and reading the frontier in the next iteration is $|V|$ linear reads. Under this model, for an average degree of 8 we would expect 25 percent of accesses to be random and 75 percent to be linear.

Based on these numbers, connected components has approximately half the random accesses of breadth-first search, but neither is 100 percent random. Clustering coefficients, as previously stated, has nearly 100 percent linear access by construction.

Table 8: Number of data elements that can be read per second as determined by microbenchmarks.

Technology	Element Reads per Second		Ratio
	Linear	Random	
Main Memory	1,762,000,000	14,300,000	53:1
Solid State Drive	68,800,000	80,000	860:1
Hard Disk	12,500,000	500	25,000:1

Now we turn our focus to the disks that Hadoop uses for storing the input, intermediate, and output data. In the previous microbenchmark, the DDR3 DRAM was able to service a random 8-byte memory request in 69 nanoseconds. This equates to 14.3 million random memory requests per second. The best spinning hard drives can complete 500 random reads per second. Flash memory or solid state drives (SSDs) can execute 80,000 random reads per second.

If we consider the bandwidths in Table 8 in terms of latency to access a single random element, the latency of main memory is 69 nanoseconds. The latency of a hard disk is 2 milliseconds and the solid state drive is 12.5 microseconds. Little’s Law describes the relationship between latency, bandwidth, and concurrency. It states that concurrency C is equal to the product of latency L and bandwidth B .

$$C = L \cdot B \tag{11}$$

The Cray XMT exploits this relationship to cover network and memory latency with threading. The Cray XMT Threadstorm processor operates at 500 MHz and has 128 hardware streams. Using Little’s Law, we calculate that the Threadstorm processor can tolerate up to 256 nanoseconds of latency. Likewise, a 3.4 GHz desktop workstation with 8 threads can tolerate up to 2.35 nanoseconds of latency, on the order of the L2 cache latency.

5.3.2 Building a Model for Access Time

If we consider a future many-core architecture with 64 threads running at 1.0 GHz, that system can cover about 64 nanoseconds of latency to memory under the assumption that enough concurrency also exists in the memory subsystem. The same model can be applied to disk-based memories. A hard disk with 2 milliseconds of random access latency requires 2 million-way concurrency at 1.0 GHz. This number reduces to 12 thousand for solid state drives. The largest public Hadoop clusters today contain only 2,000 nodes [95].

Using the data we have collected to this point, let us consider the performance of a hypothetical future system on a large graph algorithm. We will first model connected components on a graph with 17.2 billion vertices and 137.4 billion edges. The memory footprint is given by $(|V| + 2|E|)$ 8-byte elements, or approximately 2.3 TiB. We assume the algorithm will converge in seven iterations. We do not consider network bandwidth or latency.

The time to complete a single iteration is given in Equation 12.

$$T_i = \frac{M_{ref}}{P \cdot R} \quad (12)$$

M_{ref} is the number of memory references per iteration, P is the quantity of parallelism, and R is the number of memory references per second that are supported by a single unit. The total execution time is given in Equation 13. I is the number of iterations required for completion.

$$T_{total} = \sum_{i=1}^I T_i \quad (13)$$

Under this model, each system will perform 309 billion memory references per iteration of connected components, for a total of 2.16 trillion memory reads. Let us first consider an in-memory system with 4096-way concurrency in the memory.

This is a reasonable assumption given that the 512-processor Cray XMT2 shipping today has 1024 memory controllers and four-channel memory is standard on high-end systems. If each memory channel is able to complete 15 million random references per second, the connected components algorithm will complete in 5.03 seconds per iteration, or 35.2 seconds total.

Let us consider the same algorithm running on a spinning hard drive servicing 500 random reads per second. Utilizing 65,536 disks, the time per iteration is 9,440 seconds for a total time of 66,100 seconds. Using an SSD that is 160 times faster at random reads yields a time per iteration of 59.0 seconds and a total time of 413 seconds. Both are significantly slower than the main memory system.

The previous computations assume that all reads and writes to the algorithm are to random locations. In our previous experiments and algorithmic analysis, we showed that only a fraction of accesses are random. Since disk-based storage media have higher sequential access rates, let us consider a revised model based on a more complex access pattern.

Measuring the performance of connected components in memory, we found that the processing time per edge was 8 times faster than the random access microbenchmark. For the in-memory system, if we assume 120 million references per second per memory channel (rather than 15 million), the time per iteration becomes 629 milliseconds for a total time of 4.4 seconds.

The computation time estimates for the disk-based models are more complicated. Random accesses will be charged according to the latency of random access. Linear accesses must be charged according to the linear access bandwidth. The total time will be a combination of the two costs based on the frequency of random accesses with respect to linear accesses. We assume these to be uniformly distributed regardless of system layout (This may be an unrealistic assumption. However, we will address hot-spotting and load balance later.).

The time per iteration is given in Equation 14. The latency of a unit random access is L seconds. B is the peak bandwidth of the device in bytes per second. The total number of memory references M_{ref} is divided between random and linear, as shown in Equation 15.

$$T_i = \frac{1}{C} \left(L(1 + M_{random}) + \frac{8 \cdot M_{linear}}{B} \right) \quad (14)$$

$$M_{ref} = M_{random} + M_{linear} \quad (15)$$

Our previous algorithmic analysis estimated that 11 percent of reads in connected components are random with the remaining 89 percent linear. Considering spinning hard disk drives, 11 percent of the memory references will be charged 2 milliseconds, while the remainder will be charged based on the linear access rate of 12.5 million references per second. For the graph under consideration, the computation time is reduced from 66,100 seconds to 7,270 seconds.

The solid state disk has both a high linear access bandwidth and a comparatively low random access latency. Assuming 11 percent of accesses are random, the per-iteration time is reduced to 6.56 seconds for a total time of 45.9 seconds (down from 413 seconds). All three memory types benefit from additional locality, but the main memory approach continues to out-perform hard disk and solid state drives.

In order for hard disk drives to match main memory performance, approximately 107 million drives would be needed under this model. Alternatively, the percentage of random accesses would need to be reduced to 0.04 percent. For SSDs, the concurrency required is 675,000 or a random percentage of 1.1 percent. It is unlikely that this extreme level of spatial reuse can be found in graph algorithms that have been shown to contain little locality.

Breadth-first search has a higher percentage of random accesses to the data, and the results mirror those of connected components with a higher magnitude. The

Table 9: Estimated execution of breadth-first search on a graph with 17.2 billion vertices and 137 billion edges.

	Concurrency	Predicted Execution Time (sec)	
		100% Random Access	25% Random Access
Main Memory	4,096	3.36	1.48
Solid State Drive	65,536	39.3	9.83
Hard Disk	65,536	6290	1570

test graph is an RMAT graph with 17.2 billion vertices and 137 billion edges. If we estimate 25 percent of memory accesses are random, the model indicates that the main memory system will complete in 1.48 seconds versus 9.88 seconds for the solid state disk and 1,570 seconds for the spinning hard disk.

Clustering coefficients demonstrates a significantly lower level of random access than breadth-first search or connected components. If we consider an RMAT graph with 17.2 billion vertices and 275 billion edges, the memory footprint is approximately 2.47 TiB. Projecting forward the characteristics of the data generator, we estimate the number of reads required for the triangle-finding algorithm is 191 trillion. Based on measurements from the STREAM benchmark, the Intel Core i7 is able to sustain 16.5 GiB per second, or 270 million linear memory references per second. With 4096-way concurrency in the memory, the algorithm will terminate in 173 seconds.

The fraction of disk accesses that will likely be random is equal to the sum of the vertex degrees, or the number of edges. For the graph under consideration, we estimate the percentage to be 0.144 percent. Based on this information, the model predicts that the spinning disk should complete in 8,620 seconds with 65,536 disks. The SSD, which offers 550 MiB per second sustained bandwidth, will finish in 94.8 seconds with 65,536-way parallelism. The large fraction of linear data access makes the SSD an attractive choice with a 1.82x improvement over main memory for our hypothetical system.

In the preceding analyses, we did not consider communication (latency or bandwidth) or data distribution. Communication latency between nodes is assumed to be zero and bandwidth infinite. The data accesses are assumed uniformly random across disks. The data access pattern and communication costs are likely independent of the choice of storage medium. Under these assumptions, the main memory algorithm implementations are estimated to run orders of magnitude faster than the hard disk and solid state drive implementations in all but one case (using SSDs for triangle enumeration).

For most data sets, the distribution of access is not likely to be uniform. Power-law distributions in the data will often result in power-law distributions in the access pattern. Hot-spotting is a common real effect in large graph problems. Hot-spotting limits the quantity of parallelism that can be exploited. In our algorithmic analysis of random versus linear access patterns, we assumed a uniform distribution of random and linear accesses per disk. Reading the neighbors N of a particular vertex in compressed sparse row format involve one random read and $|N|$ linear reads. If $|N|$ is subject to a power-law distribution, then some disks will have a very small number of random reads and a very large number of linear reads, while most disks will have many random reads. This causes load imbalance in the system. Execution time would be measurably longer given that random reads are much more expensive than linear reads. DRAM memory is subject to the same performance gap between random and linear reads, but at a much smaller magnitude.

One possible opportunity for hybrid operation is to map portions of data in an algorithm that are accessed randomly to devices with good random access performance (DRAM) and elements of data with repeated linear access to devices with fast linear read performance, such as SSDs. The performance of DRAM is higher than solid state drives across the board. This hypothetical system only makes sense if we consider an aggregate storage capacity that is larger than aggregate main memory,

or offers greater parallelism.

In the model, the hybrid system would perform according to the Equations 16, 17, and 18.

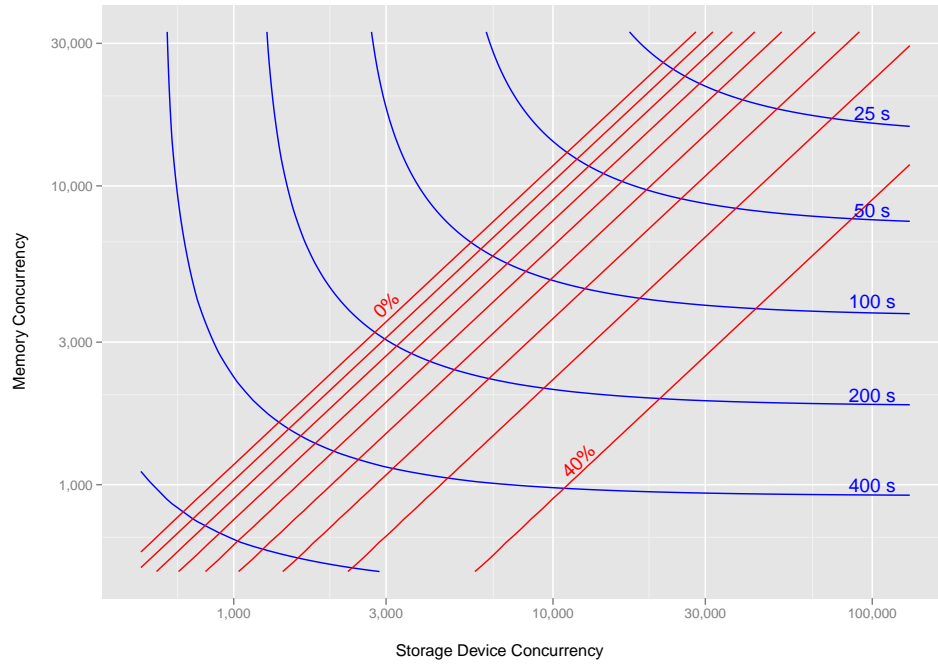
$$T_{total} = T_{linear} + T_{random} \tag{16}$$

$$T_{linear} = \frac{8M_{linear}}{B \cdot P_{linear}} \tag{17}$$

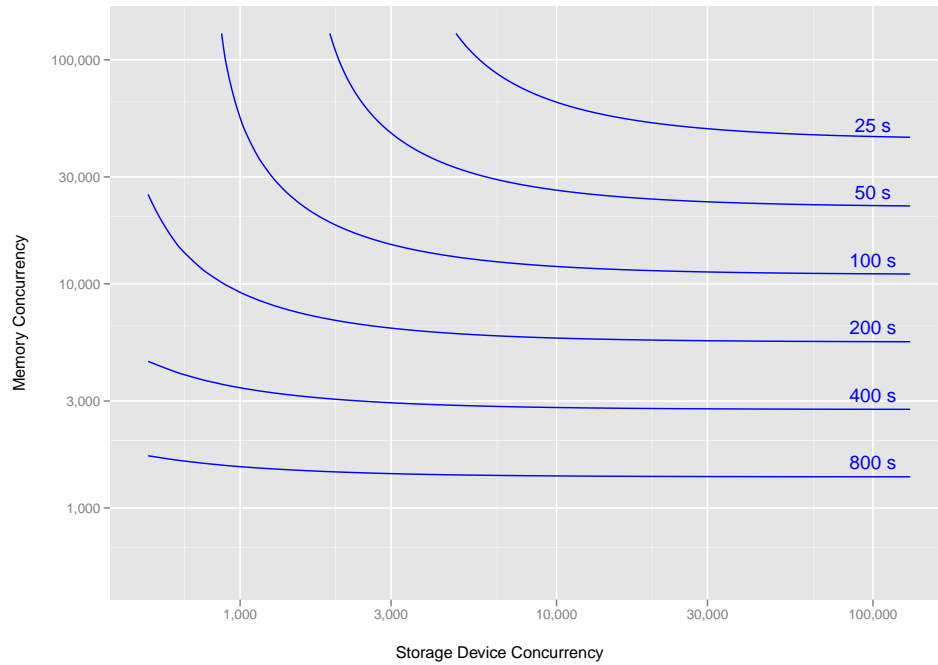
$$T_{random} = \frac{M_{random}}{R_{random} \cdot P_{random}} \tag{18}$$

For example, let us consider breadth-first search on a graph with 1.1 trillion vertices and 17.6 trillion edges. The memory footprint of the computation is approximately 158.3 TiB. With 4096 channels of memory, each channel would require 38.7 GiB. If this is feasible to build, the memory-only algorithm will complete in approximately 158 seconds. Using 65,536 solid state drives only, the algorithm should execute in 1,050 seconds. If we assume that the 25 percent of accesses that are random are stored in DRAM and the 75 percent of accesses that are linear are stored on SSD, the combined execution will be 93.1 seconds using both technologies together, an improvement of 41 percent. The full results of the model simulation are depicted in Figure 26(a).

In the figure, both the number of memory devices and the number of solid state devices are varied and performance time is measured as if randomly-accessed elements are stored in the memory devices and linearly-accessed elements are stored in the SSD. The blue lines indicate contours of iso-performance where the execution time is constant. For a given performance level, there is a marginal utility in adding memory or storage devices. For example, at 200 seconds, additional disks beyond 10,000 do not significantly reduce the number of memory devices needed. Likewise, additional memory devices beyond 10,000 do not significantly reduce the number of disks needed.



(a) Breadth-first search



(b) An algorithm with 75 percent random memory accesses

Figure 26: Iso-performance contours (blue) varying concurrency of DRAM and SSD devices in the hybrid system. Red lines indicate percentage improvement over the in-memory graph algorithm with the same memory concurrency. The input graph has 1.1 trillion vertices and 17.6 trillion edges.

Based on our measurements and models of in-memory breadth-first search execution, the red lines in the figure indicate the performance improvement of the hybrid system over a memory-only system and in-memory algorithm with the same memory concurrency. In the space to the left of the first red line, the execution time is slower than the in-memory only algorithm. To the right, the differential improves to as much as 43 percent in the lower right-hand corner.

In Figure 26(b), we present simulation model results of the same parameters on the same graph, but this time with a hypothetical algorithm in which 75 percent of data accesses are random. Note the higher execution times and flatter curves.

One challenge to exploiting this hybrid storage and execution model is that the MapReduce programming model contains no sense of locality. The intuition about memory access patterns would need to come from the programmer or algorithm developer. MapReduce also does not make any reference to the underlying data type. It is unaware whether it is working on the graph representation itself or intermediate state of the algorithm. The bulk synchronous parallel model (BSP) or Pregel seems more amenable to such a hybrid setting. State variables are likely randomly accessed, while the adjacency list is more likely (although not guaranteed) to be accessed in a predictable order.

5.3.3 Estimating Concurrency versus I/O Time

In Section 5.1, we defined the goal of a MapReduce job to maximize the computation in between input/output cycles in order to amortize the rather high cost of reading data from and writing data to disk. In this section, we will consider the concurrency required at the storage level to fulfill this criteria.

Let us define the computation time as the time estimated by the performance model for the algorithm to compute in main memory only using a concurrency of 4,096. We restrict the total I/O time to be equal to the computation time and

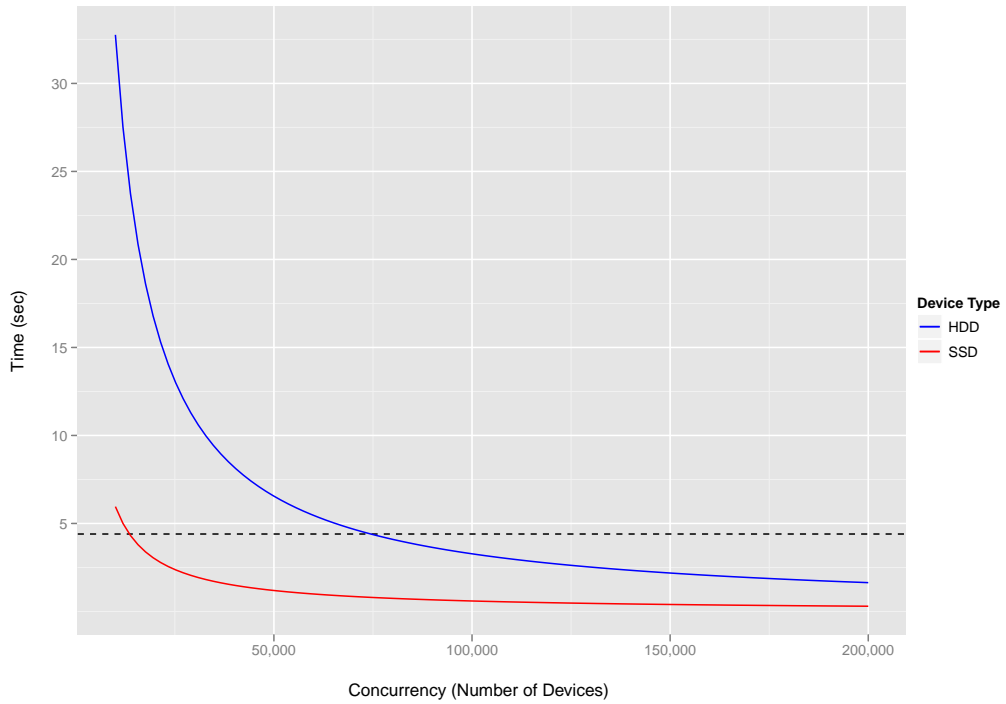
assume that the I/O time is evenly divided between input and output.

The test graph will be a graph with 17.2 billion vertices and 137.4 billion edges. For connected components, we assume 7 iterations to completion. At 120 million memory references per second (measured) per way of concurrency, the time per iteration is expected to be 629 milliseconds, with a total computation time of 4.4 seconds. The estimated I/O time as a function of concurrency is given in Figure 27(a). For a spinning disk drive with a peak linear read bandwidth of 100 MiB per second, we will perform seven I/O cycles of 2.34 TiB each. To perform these in 4.4 seconds, approximately 75,000 disks are required. Using SSDs with a peak read rate of 550 megabytes per second yields a concurrency of 13,500 SSDs required.

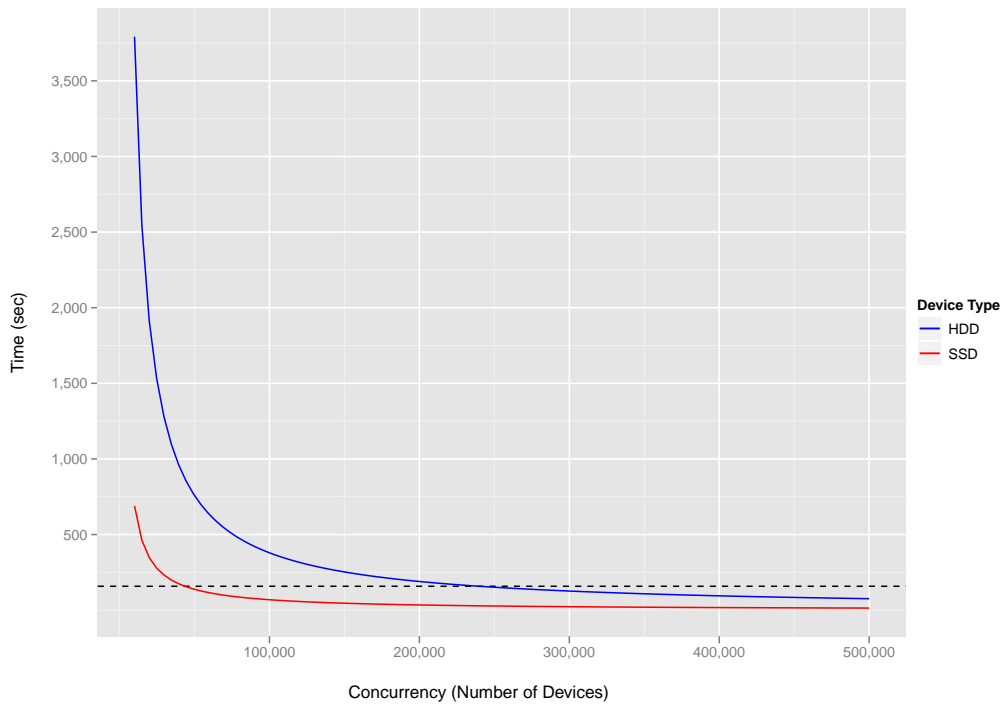
When considering breadth-first search, let us assume 12 steps will be required to traverse the entire graph. The in-memory algorithm will read the graph once and complete execution in 1.48 seconds in the hypothetical system. The disk-based systems will need to perform a series of 12 I/O operations of 1.37 TiB each whose total execution time should be equal to 1.48 seconds. To meet this requirement, approximately 225,000 hard disks or 41,000 SSDs will be needed.

On a larger graph with 1.1 trillion vertices and 17.6 trillion edges, only a modest increase in size is necessary. For breadth-first search, we again assume 12 iterations and 158 seconds to execute in memory on the hypothetical multithreaded system. Each of 12 I/O operations reading 158 TiB will be necessary. To perform all 12 I/O operations in 158 seconds, 240,000 hard disks or 44,000 SSDs will be needed. The I/O time of breadth-first search on a graph with 1.1 trillion vertices and 17.6 trillion edges is presented in Figure 27(b).

In the next chapter, we will more closely examine opportunities for mapping algorithms to architectures in a higher level, hybrid system architecture.



(a) Connected components on 17.2 billion vertices and 137.4 billion edges.



(b) Breadth-first search on 1.1 trillion vertices and 17.6 trillion edges.

Figure 27: Estimated I/O time for graph algorithms using hard disk drives and solid state drives. Dashed line indicates in-memory computation time.

CHAPTER VI

A HYBRID SYSTEM ARCHITECTURE

6.1 A Hierarchy of Data Analytics

Up to this point, we have considered massive graph analytics as running one or a small number of graph algorithms on a large graph with billions of edges. The graph may exist in memory on a large shared memory system or on disk in a large Hadoop cluster or SQL relational database system. The graph changes over time, and we have addressed algorithms for reacting to the changing graph.

A shared memory system, such as the Cray XMT, with hundreds of terabytes of main memory excelled at large graph queries. High memory bandwidth and relatively low latency (compared to disk) access to memory, when utilized by immense parallelism, contributes to fast graph analysis. In Chapter 5, we showed that the MapReduce and Bulk Synchronous Parallel programming models can easily express large graph queries, however they impose algorithmic limitations that extend beyond hardware or implementation. The shared memory programming model hides or eases these limitations.

A cluster of commodity machines and disks offers scale-out capability beyond the global shared memory that can be provided by today's high performance computing hardware. A MapReduce cluster or other large-scale data warehouse solution can scale to petabytes of storage on disk. If the graph under consideration is of this scale, and analytics must be run on the entirety of the edge set, then such a scale-out solution may be the only option.

On the other hand, if the graph (or a significant portion of the graph) can fit into 128 TiB of global shared memory, let us consider a different option for the commodity

cluster of disks. The graph is a relational abstraction of real data. This real data may be in the form of emails, transaction logs, protein sequences, or other sources of real-world experience. From this unstructured or semi-structured data, we identify actors (vertices) and look for evidence of co-action or relationship (edges). The final graph summarizes the real data. Each vertex and edge in the graph can be tied back to one or more pieces of real-world data that have been collected. In fact, multiple different graph representations can be formulated from the same set of collected data.

The raw, unstructured data is often much larger than the graph itself. Running complex analytics on the raw data is expensive and likely infeasible. We build the graph to look for more complex relationships among the simple relationships in the data. The commodity cluster has the storage capability to maintain the raw data over a long period of time, while the shared memory machine with only terabytes of memory is adequate to store the graph.

In a production environment, however, the graph is not the end, but a means to an end. Analysts, or their applications, are asking questions of the graph. These questions may occur with regular frequency or may be ad-hoc in nature. The output of one query is often the input to another or a series of further queries. The result may even motivate the analyst to go back to the raw data that the edge or edge set represents.

As much as it was infeasible to run the complex graph analytic on the raw data—so the graph representation was used—the analyst may wish to ask questions that are infeasible to run on the entirety of the graph representation. In this case, the relevant community or communities in the graph are extracted and brought to the local workstation for in-depth analysis. A workstation has only gigabytes of main memory, so the complexity of the data is reduced by orders of magnitude.

Figure 28 depicts a hierarchy of data analytics. The commodity cluster contains the largest volume of data storage and sits at the top. It holds the raw, unstructured

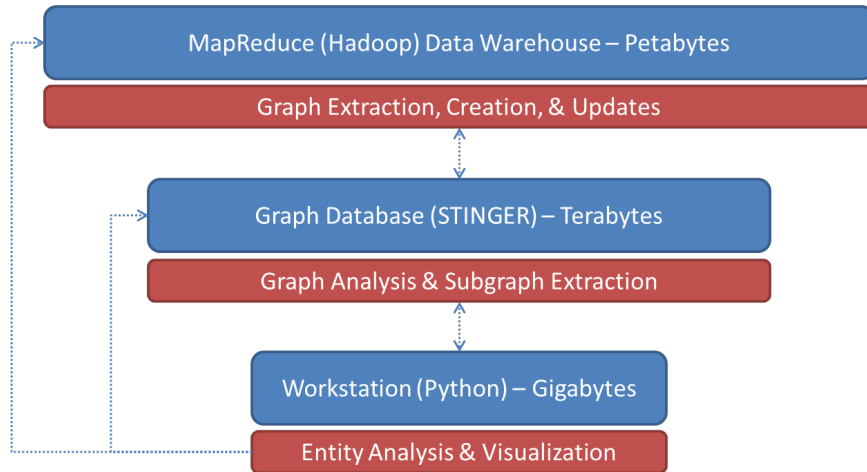


Figure 28: A proposed hierarchy of data analytics the includes raw, unstructured data in a commodity cluster, a special purpose high-performance graph representation, and many client analytics that operate in parallel.

data. Below it, a graph database is built on a shared memory platform that has an order of magnitude less memory, but is faster and more flexible. Below the graph database, the analyst workstation analyzes smaller subgraphs extracted from the graph database.

Raw, unstructured data flows into the data warehouse as it is created and collected. The data warehouse, which specializes in data parallel operations, extracts the entities and relationships from the text. These new updates are sent to the graph database as a stream of new edges. The graph database (STINGER in this example) processes these new edge updates in its internal graph representation and updates continuously running analytics. Analysts using advanced programming environments and domain specific languages, such as Python, receive subgraphs from the graph database for further investigation. It is difficult to visualize graphs with more than several hundred vertices. Subgraph extraction is essential for human-computer interaction at this step.

The inverted triangle in Figure 28 closely resembles the memory hierarchy that is common to every computer architecture textbook. In a memory hierarchy that includes cache (at the bottom), main memory (middle), and disk (top), moving from a

lower level to a higher level typically involves orders of magnitude more storage, orders of magnitude less bandwidth, and orders of magnitude greater latency or response time.

In the proposed hybrid architecture, we observe the same relationships. Only gigabytes of graph data is available to the analyst at their workstation. Computing on the graph in memory is faster than fetching additional graph vertices and edges from the graph database. The graph database, however, contains terabytes of data. It holds all or most of the graph vertices and edges. The graph database represents or summarizes the raw data held in the data warehouse. The cost to get additional data out of the warehouse is higher than out of the graph database directly.

The hierarchy of graph systems and representations behaves much like a memory hierarchy, and we can apply many well-understood aspects of the memory hierarchy to this problem. All data that resides in a lower level of the hierarchy must also reside in all higher levels. The data that an edge represents in the graph database must remain in the data warehouse. A strategy for good performance throughout the system is to keep the most relevant data in the graph database so that a minimum number of queries must require intervention by the data warehouse.

There are two notable exceptions to the memory hierarchy analogy. The first is that our hierarchy of data analytics is read-only. While data can move from the data warehouse to the graph database (or even multiple graph databases), the graph database will not modify the raw data. Likewise, a client analytic cannot change the relational graph. Since all levels are read-only, there is no need for an analogue of cache coherence protocols, and the resulting system is likely to be much more scalable.

Second, unlike in a memory hierarchy, the data in any one level is not necessarily in the same representation in other levels of the hierarchy. The data in the data warehouse is likely text or other raw, unstructured data. The text is transformed into a graph representation in the graph database. The client analytic may transform

the graph into any other representation that is necessary. This implies an additional level of logic that must exist at each level of the data hierarchy.

In the *massive streaming data analytics model*, the graph is an infinite stream of edges, each created by a particular event or interaction. Given a particular hardware platform, we store as much of the edge set in memory as possible. Our analysis takes place on this edge set, as well as the new edges that appear in the stream in the future.

When the data representation becomes full, the stream of edges will not cease. Rather, new edges will need to be evaluated and stored, if relevant. Inserting a new edge into the representation requires other data to be removed and overwritten. One possibility is to purge the oldest data in the graph. Other approaches may utilize analytic results to determine the data least used (or containing the least information).

A similar challenge exists in the hierarchy of data analytics. The graph server contains a representation (or summarization) of the data in the warehouse. If a client analytic requests vertices or edges not in the graph representation, they must be extracted and computed from the data warehouse. The graph server will need to select vertices or edges to replace, analogous to a cache or page replacement policy.

6.2 Modeling a Hybrid System

In the proposed hybrid architecture (see Figure 28), the underlying assumption is that all of the edges in the graph, or all of the raw data that makes up the edges in the graph, resides in a disk-based storage system, such as a MapReduce cloud. A graph algorithm can be run on the data there, but is relatively expensive to run. The disk-based system can also extract edge representations for use elsewhere in the hierarchy.

Also in the proposed architecture, multiple graph databases exist. Each graph database runs on a multithreaded, shared memory system, and contains some subset

of all the graph edges. These subsets can be based on vertex label, edge type, time-stamp, or another user-specified criteria. Graph analytics can be run on the graph databases provided that they contain all of the edges necessary for the query. If edges are missing from the graph database, they must be retrieved from the disk-based system first and inserted into the graph. For the purpose of this discussion, the graph database will be based on STINGER.

The situation is similar to a relational database in which queries can be executed on the data resident on disk or an index in memory. The index in memory is faster if an index already exists that meets the requirements of the query. If such an index does not exist, the query will be run on the full table on disk, typically orders of magnitude slower.

Thus to run a generic graph algorithm on a subset of the edges and vertices on the disk-based system, first the edges are filtered out of the large graph and then the algorithm is executed. To run a generic graph algorithm on the graph database, first the missing edges must be identified and extracted from the disk-based system, inserted into the graph representation, and then the algorithm is executed.

Since the algorithm can be executed at either level of the data analytics hierarchy, it makes sense to execute it where the time to completion is the smallest. To do so requires that we evaluate both scenarios based on the size of the subgraph being computed on and the number of edges missing from the graph database that must be retrieved from the disk-based system. If only a small percentage of edges are missing and these edges can be easily extracted and inserted into the graph database, it should require less time to run on the multithreaded graph database than to run on the disk-based system. However if the entire graph must be transferred, and the transfer time is large compared to the execution time, it may be more efficient to compute directly on the disk-based system.

The trade-off depends on many factors including the size of the disk-based system,

Table 10: Experimental parameters

Parameter	Value
Number of memory units	4,096
Number of storage devices	65,536
Graph database (per unit values)	
Updates per second	2,400,000
BFS references per second	34,000,000
Connected components references per second	120,000,000
Triangle counting references per second	270,000,000
Disk-based cloud (per unit values)	
HDD linear references per second	12,500,000
HDD random references per second	500
SSD linear references per second	68,800,000
SSD random references per second	80,000

the speed of the disks, the size of the multithreaded system, the speed of the memory, the insertion update rate of the data structure, and others. We will use the model described in previous sections to estimate the algorithmic execution time given these parameters and the size of the data.

6.2.1 Experimental Method

We will examine popular static graph algorithms connected components, breadth-first search, and triangle counting. These algorithms display a variety of characteristics found in other algorithms.

We fix several machine parameters so as to consider fewer variables in our simulations (see Table 10). The disk-based system will contain 65,536 disks that can be hard disk drives (HDDs) or solid state drives (SSDs). The graph database will be modeled by a shared memory system with 4,096-way concurrency in the memory. STINGER on this system will support 10 billion new insertions per second. STINGER today

can support 100 million insertions per second if the edges are sorted, packed, partitioned by source vertex, and unique. A disk-based system, such as Hadoop, can easily restructure the edge list in this way. A STINGER insertion rate of 10 billion edges per second is equivalent to 100 current Intel x86 systems. For the purpose of this experiment, we do not consider network bandwidth or transfer time since network communication can be overlapped with graph update and modification.

We make the conservative assumption that the disk-based system can extract the queried edges directly, without reading the entire edge list, and that these edges are stored contiguously on disk. This is a valid assumption if edges are being stored and extracted by vertex identifier or by edge type. In these cases, $O(|V|)$ random access latencies would be added to account for identifying the edge blocks of interest, increasing the time to extract the edges.

The time to compute the graph algorithm on the disk-based system will consist of two parts: 1) time to extract the subgraph edges from the dataset and 2) time to compute the algorithm on the subgraph. The time to compute the graph algorithm on the graph database will consist of three parts: 1) time to extract a fraction of the subgraph edges from the disk-based system, 2) time to update the graph data structure, and 3) time to compute the algorithm on the subgraph in memory. The objective is to compute the trade-off between fraction of edges that must be retrieved and the cost to move the edges into the graph database.

6.2.2 Results

Let us consider connected components on a graph with 2^{40} or 1.1 trillion vertices, and 17.6 trillion edges that takes 7 iterations to complete. The number of memory references is approximately 261.7 trillion. By algorithmic analysis, the percentage of random accesses is approximately 6 percent.

Each hard disk in the disk-based cloud can sustain 500 random references per

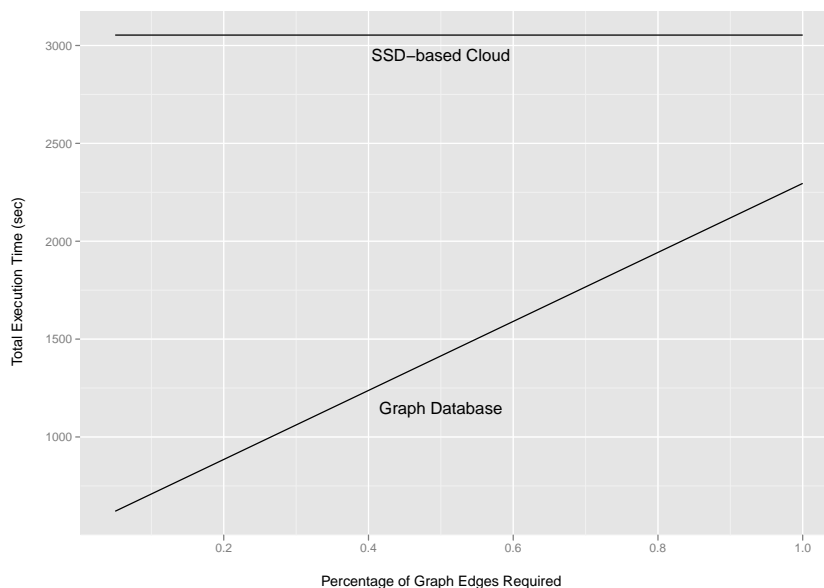


Figure 29: Connected components execution time comparison for a hybrid system with a shared memory graph and SSD-based storage system. The input data is an undirected graph with 1.1 trillion vertices and 17.6 trillion edges.

second and 12.5 million linear references per second. On a disk-based system with 65,536 spinning hard disk drives, the time to extract the full graph edge list is modeled to be 21.5 seconds, and the connected components algorithm takes 133.2 hours.

The multithreaded graph database system processes 120 million references per second on connected components (as measured, taking into account the ratio of random to linear access) per memory unit. With 4096 memory units, the connected components algorithm is modeled to execute in 532.4 seconds.

In Figure 29, the percentage of edges that are missing from the graph database is varied from 5 percent to 100 percent. The time required to retrieve these edges from the disk-based cloud and insert them into the graph database varies based on the number of edges.

Under these conditions, the total time to retrieve the graph edges, insert them into the database, and process the connected components is less than 2350 seconds

on the multithreaded graph database. The disk-based system requires 133 hours end-to-end. For connected components on these two systems, it is always more efficient to extract the edge set from the disk-based system and run the graph algorithm in the multithreaded graph database.

If the disk-based cloud system utilizes SSDs, rather than spinning hard disk drives, the time to compute in the cloud is greatly reduced. SSDs can sustain 80,000 random references per second and 68.8 million linear references per second. Graph extraction is calculated to be 3.9 seconds and the algorithm runs in 3050 seconds.

The multithreaded system also takes advantage of faster graph extraction with end-to-end completion times no more than 2300 seconds for 100 percent missing edges. In this case also, the connected components algorithm runs most efficiently on the multithreaded graph database, although it is only 25 percent faster than the SSD system alone.

Turning to breadth-first search, we consider the same input graph with 1.1 trillion vertices and 17.6 trillion edges. The number of memory references is 22.0 trillion. By algorithmic analysis, 25 percent of references are accessed randomly.

On a disk-based system with 65,536 spinning hard disk drives, the time to extract the full graph edge list is 21.5 seconds, and the breadth-first search takes 46.6 hours.

The multithreaded graph database system processes 34 million references per second on breadth-first search (as measured, taking into account the ratio of random to linear access) per memory unit. With 4096 memory units, the breadth-first search algorithm is modeled to execute in 157.9 seconds.

In Figure 30, the percentage of edges that are missing from the graph database is varied from 5 percent to 100 percent. Under these conditions, the time required to extract the full graph from the disk-based system, load it into the graph database, and compute the breadth-first search is 1939 seconds, compared with 46.6 hours to compute the entire workflow on the disk-based system.

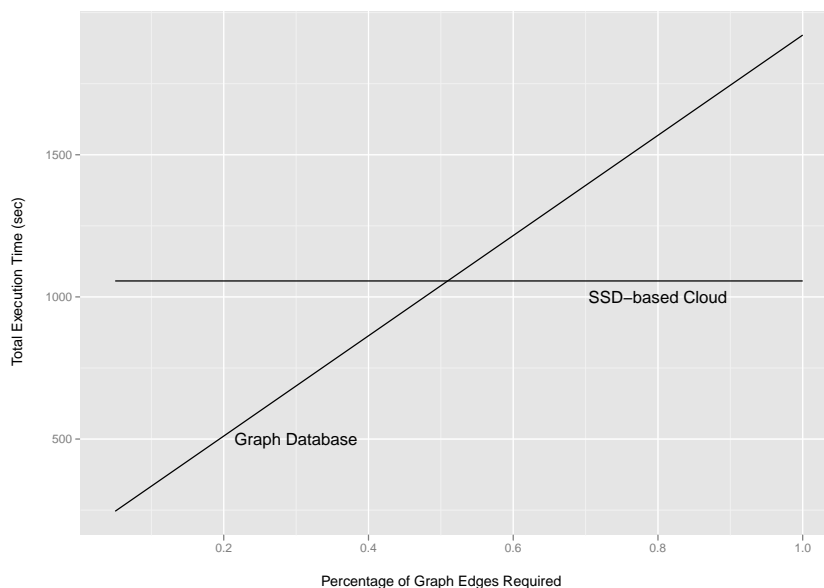


Figure 30: Breadth-first search execution time comparison for a hybrid system with a shared memory graph and SSD-based storage system. The input data is an undirected graph with 1.1 trillion vertices and 17.6 trillion edges.

If the disk-based cloud system utilizes SSDs, rather than spinning hard disk drives, the time to compute in the cloud is greatly reduced. Graph extraction takes 3.9 seconds and the algorithm runs in 1056 seconds on the SSDs alone.

The multithreaded system takes advantage of the SSD-based cloud with an end-to-end completion time no greater than 1921 seconds for 100 percent missing edges. When the fraction of edges that are missing from the graph database is 50 percent or less, it is advantageous to extract these edges from the cloud and run the breadth-first search on the graph database. Otherwise, it is faster to run the algorithm where the data is in the cloud. The dominant cost in this model is the time to update the graph database with the missing edges.

Triangle counting has a significantly lower percentage of random accesses than connected components or breadth-first search. We consider an input graph with 2^{32} or 4.3 billion vertices and 68.7 billion edges. We estimate the number of memory references to be approximately 191 trillion. The fraction of these accesses that are

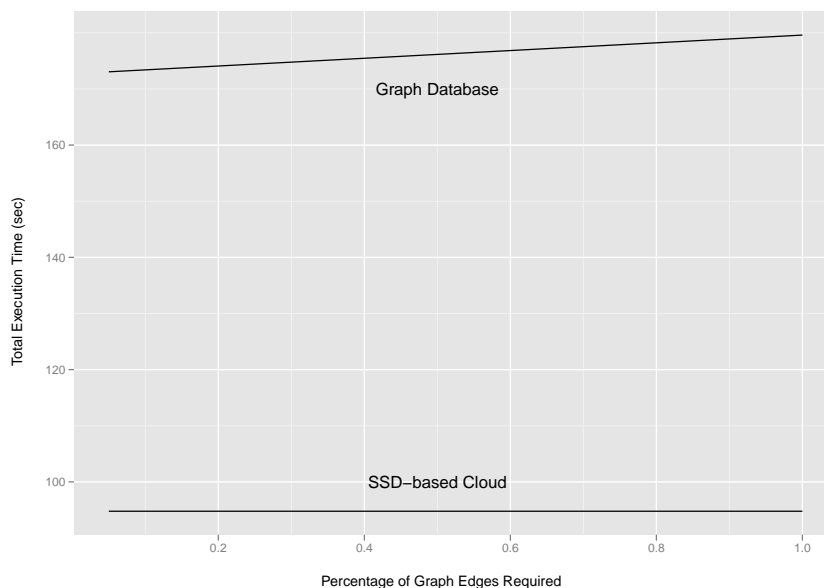


Figure 31: Triangle counting execution time comparison for a hybrid system with a shared memory graph and SSD-based storage system. The input data is an undirected graph with 4.3 billion vertices and 68.7 billion edges.

random is selected to be 0.144 percent based on an algorithmic analysis.

On a disk-based system with 65,536 spinning hard disk drives, the time to extract the full graph edge list is 85 milliseconds, and the triangle counting algorithm takes 8626 seconds.

The multithreaded graph database system processes 270 million references per second on triangle counting (as measured, taking into account the ratio of random to linear access) per memory unit. With 4096 memory units, the triangle counting algorithm is modeled to complete in 172.7 seconds.

In Figure 31, the percentage of edges that are missing from the graph database is varied. Under these conditions, the time required to extract the full graph from the disk-based system, load the edges into the graph database, and run triangle counting is less than 180 seconds, compared with 8626 seconds to compute on the disk-based system.

Switching the disk-based cloud to SSDs, graph extraction time is reduced 15

milliseconds. Triangle counting run in 94.8 seconds on the SSD-based system. The multithreaded system completes in 173.0 to 179.6 seconds, depending on the fraction of edges missing from the graph database. In this scenario, the SSD-based system excels at reading the largely linear edge lists, and is the more efficient choice regardless of the number of missing edges in the graph database. However, the disk-based system is only 45 percent faster than the multithreaded graph database, so either is a realistic option in a production environment.

6.2.3 Discussion

It is important to consider both fraction of accesses that are random, as well as the total quantity of random accesses. In Figure 29, connected components is best run on the multithreaded in-memory graph database regardless of the fraction of edges that must be retrieved from the graph on disk. Despite the fact that connected components algorithm is over 94 percent linear access, the graph database is faster than the disk-based cloud.

Breadth-first search, on the other hand, is 25 percent random access, but demonstrates a clear trade-off in Figure 30. The cost of transferring edges from the cloud to the graph database is such that it is faster to run the breadth-first search in the cloud if more than half of the edges must be retrieved.

Looking only at the fraction of random access, it would seem that the results should be inverted. The multithreaded system should excel at breadth-first search, while the disk-based system might be better at connected components. However, if we consider the total quantity of random and linear accesses, we can better understand the modeled results.

The total quantity of linear and random accesses for the two algorithms on the same graph is compared in Figure 32. Because connected components must make seven complete passes over the edge set, the total number of references is an order of

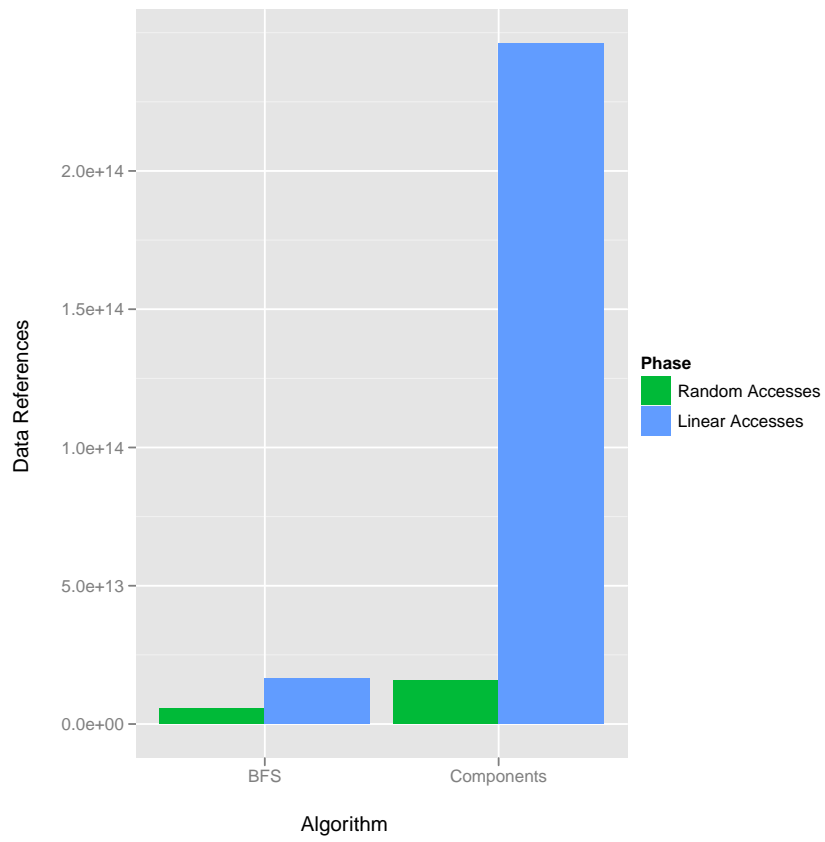


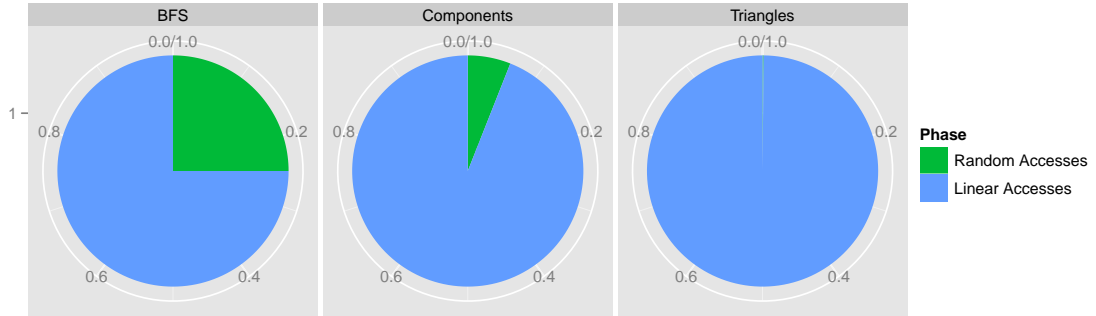
Figure 32: The total number of random and linear data references by algorithm. The input graph has 17.6 trillion edges.

magnitude higher than breadth-first search, which makes a single pass over the edge set. As a result, the 6 percent of accesses that are random for connected components is 15.7 trillion, compared to the 5.5 trillion for breadth-first search. For a single device, main memory is approximately 180 times faster for random access than an SSD. Our hypothetical system had 16 times more SSDs than memory units, so main memory is still 11 times faster than SSDs in aggregate for random access. As the absolute number of random accesses increases, the multithreaded in-memory system gains a greater advantage over the on-disk system.

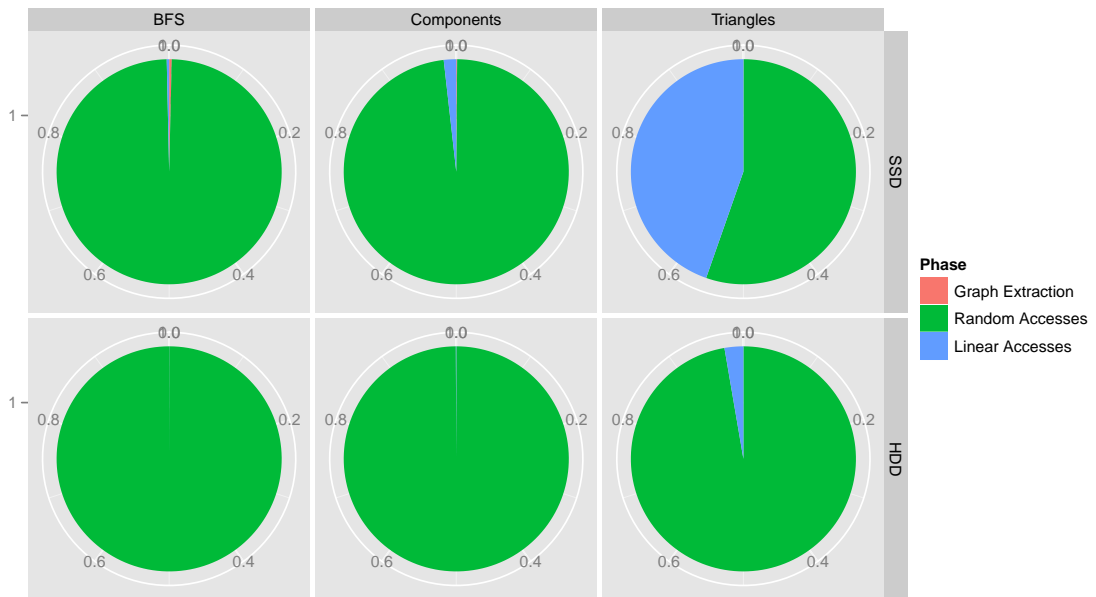
Considering linear access rates, a single main memory unit is only 25 times faster than a single SSD. Including a factor of 16 more SSDs, the aggregate performance for linear access is nearly equal for in-memory computation or on-disk. On the system level, it suffices to consider random access as the dominant factor in determining where to run the computation. Next we will show the effect of random access on modeled execution time.

When an algorithm combines random and linear access patterns, the high cost of random access, regardless of the storage medium, is the dominant factor. Consider that a spinning hard disk drive can fetch 25,000 linear elements for every random element. The solid state disk has a better ratio at 860 to 1, and DRAM is closer to 50 to 1.

The impact can be seen more clearly when considering the quantity and fraction of execution time for linear and random access to the data. Figure 33(a) contains an algorithmic analysis of breadth-first search, connected components, and triangle counting. As previously discussed, for a graph with an average degree of 16, breadth-first search requires approximately 25 percent of data references to be random. Six percent of accesses will be random in connected components. For triangle counting, approximately 0.144 percent of accesses are random. The blue portion in Figure 33(a) represents linear accesses to the data, while the green portion represents random



(a) Fraction of data references that are random or linear accesses



(b) Fraction of modeled execution time

Figure 33: Algorithm modeling random and linear access by quantity and time for hard disk drives and solid state disks. Breadth-first search and connected components are modeled for a graph with 1.1 trillion vertices and 17.6 trillion edges. Triangle counting is modeled for a graph with 4.3 billion vertices and 68.7 billion edges.

references.

In Figure 33(b), we plot the fraction of modeled execution time consumed by each phase of the algorithm. The time to extract an adjacency list from the edge data is shown in red. The green portion represents the time spent performing random accesses to the data, while the blue portion is the fraction of time spent performing linear accesses.

Comparing the two figures, we observe that the majority of accesses to the data structure are linear in nature (reading the adjacency list of a vertex in order). Storage devices are tuned for this access pattern and perform the operations quickly. In the lower figure, however, the overwhelming majority of execution time is consumed by random access. For connected components, the SSD-based system spends 97 percent of its time handling the 6 percent of data references that are random. For triangle counting, 0.144 percent of accesses take up 55 percent of the execution time.

The large discrepancy in random and linear access rates results in a pareto distribution in execution time. Algorithms with extreme locality are still dominated by the random access factor. Increasing the speed of random access in a storage device will greatly accelerate all algorithms, even while holding linear access speed constant.

CHAPTER VII

RISE OF THE MACROARCHITECTURE

A significant portion of computer architecture research focuses on microarchitecture. The goal of this research is to improve single-threaded performance by identifying opportunities for instruction-level parallelism (ILP) and branch prediction, improve cache effectiveness through different allocation schemes and cache arrangements, and increase throughput using various techniques such as vectorization or VLIW. In parallel computing, microarchitecture often looks at efficient protocols for communication and sharing between cores on a chip and between sockets.

The data-intensive computing revolution should cause us to re-think computer architecture in a new light. Data is larger than ever before, and multiple data representations are needed for different analytics. Large graph and relational analytics can benefit from a shared memory view of the data. Data parallel tasks, such as indexing, are well-suited to distributed or shared-nothing systems. Algorithms with large volumes of random data access run faster in DRAM technologies. By contrast, those algorithms that read data once in sequence benefit from linear access devices such as SSDs. Any number of client applications may contain both types of queries at multiple levels of granularity. The application is no longer contained in one system, but is a collection of applications running on a variety of systems.

In this new world of massive data analytics, there is a great need for research in computer *macroarchitecture*. Queries must be mapped to the correct data representation running on the appropriate system. Data representations must be in sync, if only loosely. The data representations themselves must intelligently respond to requests and adapt as the query environment changes.

In this chapter, we will discuss conclusions and recommendations for computer architects that are motivated by the preceding research in order to better cope with current and future trends in massive graph analysis.

7.1 *Parallelism*

At the scale of data under study (trillions of edges), parallelism is needed throughout the system. From the programmer's perspective, parallelism can take many forms, including loop-level and task-level. In Chapter 3, parallelism played a key role in enabling large graph algorithms on the Cray XMT. Exposing a fine granularity of parallelism to the compiler (while avoiding hot-spotting and synchronization) gives good performance and scalability on today's massively parallel systems. Systems of the future will support more threads and more opportunities for parallelism will be needed from the algorithms.

At the runtime perspective, parallelism must be thought of with a short lifetime. Work items in a parallel graph algorithm have very short lifespans with small footprints. The quantity of parallelism can change drastically depending on the data, the loop, and the progress through the computation. Thread creation must be lightweight. Thread execution is often conditional or speculative. A runtime that efficiently manages large numbers of threads (hundreds to thousands per socket) and the changing conditions of the system will better support an irregular application.

Multicore processors have been prevalent for many years now and will likely continue to be the mainstay of high performance computing. Increasing core count and off-chip bandwidth will enable greater parallelism in graph applications. Experimental results in Chapter 4 demonstrate the capability of multi-socket multicore systems on graph computations. However, scalability of synchronization outside of a single socket is still challenging. Future systems should provide synchronization facilities that have low costs when contention is low or nonexistent.

Parallelism must exist at the level of memory and disk. A combination of parallelism in compute and storage is necessary for latency tolerance. Experimental results in Chapter 5 highlight the need to distribute data access patterns to maximize bandwidth, both for random and linear access patterns. The immense size of data under study requires such an approach. The balance of parallelism at the memory, disk, and processor must be tuned so that parallelism can overcome latency.

Multiple applications connecting to and reading from the graph data store exhibits a different kind of parallelism. The graph data store must be able to service these requests with low latency while also staying up to date with new data. There may be multiple graph data stores representing different aspects of the raw, unstructured data held in the warehouse. If overlaps in the queries or graphs can be discovered, there exists opportunities to tune for these so long as the lifetime of the overlap is significantly greater than the cost to optimize.

7.2 Shared Memory and BSP

In Chapter 5, we explored the algorithmic trade-offs in graph algorithms between the shared memory programming model and the bulk synchronous parallel, or BSP, model employed by Pregel and other implementations. The BSP model is used throughout scientific computing on distributed memory systems that often exhibit regular computation and communication patterns. We showed that several popular shared memory graph algorithms could be easily adapted to the model.

There is a clear trade-off between program efficiency in the shared memory model and ease of programming in the BSP model. In the three experimental cases considered, the quantity of work (as measured by number of iterations, reads, and writes) was higher for the BSP algorithms. For breadth-first search, the discrepancy was modest and performance reflected this. Connected components, which should have

benefited from computing only those vertices that required it, took longer to converge from lack of communication. Triangle enumeration suffered from an explosion of intermediate data.

The criticism of shared memory programming stems from its implicit communication pattern. The programmer has less control over data movement. Understanding synchronization is more difficult and deadlock is possible. The BSP model regularizes and limits communication so as to prevent these situations and makes messaging explicit to the programmer. In the experiments, the cost of doing so is a factor of two or less as long as the size of data can be contained.

Shared memory has the distinct advantage for graph algorithms in that data (and changes to data) can flow more freely through the algorithm. Asynchrony of computation and communication creates complexity, but also opportunities for algorithms to converge more quickly. In a shared memory environment, there is no notion of place or locality, so the data does not have to be partitioned or assigned to workers based on its position. MapReduce attempts to achieve the same objective by moving work to the data, but can fail when load imbalance prevents it. Shared memory programming is agnostic. Load balance is still a concern in shared memory programming, but can be handled by the programmer (or hopefully in the future by an intelligent runtime).

Bulk synchronous parallel programming has proven itself adept in massive dense linear algebra computations. In certain cases, such as breadth-first search and connected components, BSP can be a powerful tool for graph algorithms when implemented correctly. Shared memory is likely to remain the primary programming environment for the general graph algorithm.

7.3 Latency-oriented Analytics

Chapter 4 describes an application area in which the graph is a dynamical data structure constantly in motion. Algorithms must strive to keep up with the data.

We demonstrated a data structure (STINGER) that is capable of reacting to change quickly while maintaining performance of edge insertions, deletions, and accesses in a small-world network.

Successful algorithms compute the minimum data that must be re-analyzed and recomputed given a set of changes. This is likely to be a small neighborhood around an edge or a vertex. Parallelism can be found by performing a batch of updates at one time. There are opportunities within a batch to amortize computation.

In some cases, deletions create special situations that are not easily resolvable. We proposed several approaches, including heuristics, for dealing with these cases. With heuristics, periodic recomputation is necessary, although may be deferred for hours or days within an acceptable error limit.

The goal of latency-oriented analytics is to keep analytics up to date with a streaming graph. Some static graph analytics require minutes or hours to compute once. In a streaming graph, the time between update of the graph and update of the metric should be minimized to minutes or even seconds. Data representations for these applications must support fast insertions, deletions, updates, as well as accesses by many analytics. They must be built for parallelism at the foundation.

To accomplish these objectives, STINGER creates parallelism at many levels, including multiple pathways to access the data depending on objective. The data is highly semantic with labels on vertices and edges, timestamps, weights, counts, and other metadata. The data is broken into smaller chunks that can be easily updated and moved around the system, but large enough to expose a small degree of parallelism.

Architecture should move to reflect this change in data representations. The fundamental unit of data is no longer a word or vector, but a block of data. Elements within the block reflect the status of other elements in the block. For example, the edge count is a count of the number of valid edges in the block, where a valid edge is

a 4-tuple in which the first field (the neighbor) is non-negative. Today, this metadata is kept up to date with complex synchronization protocols. A programmable memory controller could easily keep the metadata up to date on behalf of the application with no need for synchronization.

Combining the knowledge of the objectives of the graph access pattern with the layout of the data representation, a programmable memory controller could return graph edges in a format suitable for the cache hierarchy and the application. If a loop is only interested in the neighbor IDs (a basic traversal), the data should be returned to the process from the memory in “struct of arrays” form. On the other hand, if the application is reading edges to determine those with a given timestamp, returning the data in “array of structs” form puts the time information on the same cache line as the neighbor identifier. This is a level of application and data intelligence we have never seen in computing.

7.4 Memory Technologies

The analytical results of Chapter 5 demonstrate a significant gap between the access rate of DRAM technology (both linear and random access) and non-volatile storage devices. To overcome this gap and reach acceptable performance levels with non-volatile storage requires several orders of magnitude more disks and parallelism. Spinning hard disk drives offer slower linear performance than the random access performance of DRAM. For this reason, spinning hard disks should only be considered for capacity reasons.

For big data applications, system architects ought to offer large capacity DRAM technology, even if the memory is physically distributed. Parallelism in the memory is key. Multiple processing elements connect to multiple memory controllers. Memory controllers with multiple memory channels and memory banks offer additional parallelism. A modest system with 32 GiB per channel and 4096-way parallelism in

the memory subsystem would offer 140 aggregate TiB of in-memory storage. Each byte is accessible thanks to high-speed random access.

A comparable system based on solid state drives would require 500,000 or more devices to offer similar performance. While a 512-node system could offer 4096-way parallelism in the memory, accommodating 500,000 storage devices would necessitate thousands of nodes with reasonable limits on the number of devices per node. Total aggregate storage would increase using SSDs due to their underutilization, but the memory footprint of the computation could not increase without a proportional increase in execution time.

Hybrid memory technologies are a promising area of research for system designers. Identifying linear access regions in program execution or data storage and assigning these regions to devices that excel at linear access, while the remaining random accesses utilize DRAM, can help alleviate the bottleneck when the computation cannot fit entirely into memory.

Future memory technologies may also be useful. 3D memory stacking may increase parallelism within the memory while simultaneously reducing access latency. Phase change memory is non-volatile, several hundred times faster than flash, and can be built in high densities. Phase change memory could replace DRAM in future big data systems, or act in hybrid with DRAM as part of the memory hierarchy offering much larger capacities per node. Future memory technologies will consume less power and space than current storage technologies [96], and should be a candidate for adoption close to the processor.

7.5 Reliability & Resilience

As systems grow larger, the probability of component failure grows exponentially. Scientific applications manage failure through checkpoint and restart mechanisms,

including distributed and asynchronous protocols [97, 98, 99]. Hadoop, and MapReduce, handle failure through redundancy of data and computation. There is a significant overhead to detection and restart of failed computation, as well as the need to periodically re-balance data.

In Chapter 5, we observed the possibility that a BSP algorithm would produce a large quantity of intermediate data (much more than the input size). BSP algorithms, unlike pure MapReduce, maintain state between supersteps, and this state must be protected in a resilient manner. Storing larger quantities of temporary or dynamic data in a redundant fashion stresses interconnection networks and wastes accesses to the data, which are the limiting resource to performance.

Resilience can be managed in hardware with lower overhead. Triple modular redundancy, or related techniques, can handle node failure with similar system complexity without the need to re-balance data or store intermediate state to non-volatile storage. The computations of interest are extremely large and the data contain many sources of error. Care should be taken to minimize the impact of hardware failure below the noise floor of the data.

Since the graph database server contains a representation of raw data in the data warehouse, failure at this level can be handled by re-creating the lost portions of the graph from the original data. The data warehouse is better equipped to handle redundancy. The graph database can be re-built in an online fashion without stopping running analytics. When the data is back up to date, online analytics will adjust their outputs.

7.6 *Integration*

A major driver behind the adoption of MapReduce and Hadoop is their reliance on commodity hardware: servers, storage, networks, and programming environments. The economic cost to develop on top of open source frameworks with commodity

hardware is thought to be much less than application-specific custom hardware and software. At each layer (software, runtime, hardware), the gap between custom performance and commodity performance compounds. Comparing to others' work, we have obtained 100x or greater performance using custom hardware and software on problem instances of 100 times greater size [91, 100].

There is promising research using commodity microprocessors to do latency-tolerant graph analytics by using extreme volumes of lightweight threading [101]. In these experiments, thousands of extremely lightweight threads are spawned per core. Cores explicitly manage transactions in main memory enabling lightweight locking and atomic operations. This is a promising avenue of research, but also highlights the inadequacy of interconnection networks and current system design.

If we look at the system architecture of a commodity multicore server today, it bears a striking resemblance to the architecture of the first personal computers. We use a stored program and data in memory. Data is brought from memory across the bus to the CPU registers. Data in registers is computed and stored to other registers, then copied back to memory. Multicore continues to adopt this model.

The rise of multicore computing was the result of inexpensive transistors, Moore's Law, and the diminishing returns of microarchitectural innovation. Moore's Law continues to shrink digital designs, but the static system architecture prevents these new transistors from appearing anywhere but the processor (and largely in the form of ever growing caches).

Why not employ inexpensive processing units throughout the system? As far back as the 1970s, the von Neumann architecture's separation of CPU and memory was seen as a bottleneck that inhibited more creative designs and programming constructs [102]. Even though the bus between memory, the processor, and everything else has been replaced with more modern networks, bandwidth and latency to memory remain the most critical factors to performance on data-intensive problems. We

have observed experimentally that some locality exists within these problems, but not enough for caches to be effective at any size less than the problem size.

Multicore processors and parallel programming have done little to alleviate the bottleneck since the CPUs are starved for data. Placing general purpose processors on the other side of the network increases the flexibility of the system, but also increases programming complexity. A processing element closer to the data could handle operations on behalf of the CPU, including atomic operations, data movement and copying, data filtering and restructuring, and data-dependent pre-fetching. Two cores on either side of the network have a symbiotic relationship as each handles the tasks it is best positioned for.

Since the MapReduce framework, and arguably big data analytics in general, depends on massive data stores of non-volatile storage, the same processing elements can be deployed to the storage devices to handle in-place operations, indexing, search, and filtering. Netezza developed a database platform that utilized FPGAs on hard drives, but the technology has yet to be widely adopted [103].

If high performance computing and big data analytics are to depend on commodity hardware and software, then we must begin to innovate in the commodity design space. Change at the system level bears less risk than at the microprocessor level and can be prototyped as middleware in early adoption phases. New memory technologies that will be production-ready in the coming years present an excellent opportunity to make changes to the system architecture. By re-thinking and innovating the macroarchitecture, we can fully take advantage of these new technologies as they become available.

CHAPTER VIII

CONCLUSION

The volume of data produced by the digital world continues to grow. Advances in digital storage techniques enable collection and retention of the data, but do little to produce insights into meaning. Knowledge discovery requires new algorithms that solve for unknown phenomena. Many existing algorithms are unscalable to large data volumes or contain optimization problems that are NP-complete or NP-hard.

The objective of this research is to study techniques for engineering parallel graph algorithms that scale on current- and future-generation systems and data sets. We investigate architectural features and requirements to understand the interplay of data, hardware, and algorithm. With increased understanding of design trade-offs, we close the loop between algorithm and hardware design for data-intensive applications.

The key challenges to high performance scalability in static graph algorithms are parallelism, synchronization, and hot-spotting. We compute the connected components of a scale-free network with over 17 billion edges in 86.6 seconds on the 128-processor Cray XMT. This result was only possible after identifying hotspots in the shared memory algorithm. The algorithm contained an optimization that reduced the number of iterations to convergence, but the scale-free nature of the data combined with the label pushing optimization caused a hotspot to materialize at high thread counts. Removing the optimization greatly increases scalability to 113x on 128 processors. Without the optimization, however, convergence is slow on graphs with large diameters. The efficient algorithm identifies that the graph diameter is large and switches the optimization on automatically.

Realizing that the graph data under study contains noise and missing edges,

we sought to make graph algorithms more robust against changes when developing k -betweenness centrality. Traditional betweenness centrality executes successive breadth-first searches to determine shortest paths. k -Betweenness centrality augments this metric with short paths that are within k hops of the shortest path distance. We demonstrate several techniques, including reducing synchronization by restructuring the algorithm and manually unrolling loops and functions, that enable near-linear scaling up to 96 processors on the Cray XMT. This level of scalability reduces the computation from over five hours to just four minutes on a scale-free graph with 135 million edges. Running this new algorithm on a real data set, we observe that a striking number of vertices lie within two hops of the shortest path, but are not considered by betweenness centrality.

Dynamic, streaming graph analysis has many of the same challenges as static graph analysis. In streaming graph algorithms, the quantity of parallelism is greatly reduced to the size of the batch of edges being considered and often the neighborhood around them. The goal of efficient streaming graph analysis is to determine an algorithm that computes only the changed values in the graph with minimum data access.

We determined in early work that a single edge insertion does not produce enough parallelism in a large graph to fully utilize a current multithreaded system. More parallelism is needed, and can be found by considering a batch of edge insertions at a time. A side effect of this approach is per-edge work may overlap, increasing efficiency.

A data structure for streaming graphs must be able to store graph vertices and edges that contain metadata, such as degree, label, type, and time information. The graph is scale-free with a power-law distribution in the number of neighbors a vertex has. The data structure must be able to support vertices with high and low degree with minimum waste. A capable data structure will support efficient algorithms for

edge and vertex insertion, deletion, modification, and query in which the performance cost for each is predictable and similar.

We parallelize edge insertion and removal in STINGER by analyzing a series of techniques that includes batching of edge updates and a thread-safe insertion protocol that utilizes lightweight synchronization to allow multiple threads to operate on edges incident on a common vertex. This work brought the insertion performance of STINGER on the Cray XMT from 950 updates per second to over 3.1 million updates per second. On an Intel x86 server, insertion performance increased from 12,000 updates per second to 1.8 million updates per second. We demonstrate that linear scalability can be achieved in a parallel streaming graph data structure by increasing parallelism and decreasing contention.

STINGER is the basis for new streaming algorithms to track clustering coefficients and connected components. The clustering coefficients algorithm realizes that a single edge insertion (or deletion) affects only the numerator of the clustering coefficient for the endpoint vertices of the edge and their common neighbors. Different set intersection techniques can be used depending on cache configurations and the quantity of parallelism needed. The streaming connected components algorithm transforms a batch of edges from the vertex space to the component space and quickly resolves new labels. Deletions require full recomputation in the general case, but we identify several strategies using neighbor intersection and spanning trees to reduce the number of cases that cannot easily be resolved.

Disk-based cloud systems that use various implementations of the MapReduce parallel programming model are one possible alternative to shared memory, multi-threaded systems for graph computations. We discuss several aspects of the MapReduce model that are poor fits for graph algorithms. The bulk synchronous parallel (BSP) programming model, which is used by the Pregel and Giraph frameworks for graphs, better expresses graph algorithms and can be implemented in a disk-based

cloud system. We investigate the algorithmic effects of BSP graph algorithms by measuring and comparing machine-independent characteristics of execution with shared memory graph algorithms. We find that connected components and triangle counting BSP algorithms read and write more data than the shared memory equivalents. The breadth-first search algorithm is extremely similar in BSP and shared memory, except for the representation of the frontier.

Next we sought to understand the performance trade-offs of disk-based hardware for graph computations and graphs that are stored in main memory. Using a microbenchmark, the access time per data element was measured on a current-generation system for each of the graph algorithms. For connected components and breadth-first search, the random access is dominant and time per element did not scale with graph size. Triangle counting, which contains long sections of linear accesses, displays a constant access time per element for graphs up to 2 GiB.

Using the memory access time data collected and microbenchmark data for random and sequential access of leading hard disk drives and solid state drives, we develop a performance model for data access time on future systems. Future systems are likely to consist of many parallel disks, many parallel memory banks, or both. Measuring the quantity of data accesses and the proportion of random data accesses, we can suggest the expected computation execution time for a specified number of disk or memory devices. We find that hard disk drives are not competitive, except in extreme quantities over 5 million disks. SSDs are faster, but several orders of magnitude more devices are needed to match the performance of DRAM.

Putting together the algorithms, programming models, and hardware architectures, we arrive at a proposed system architecture for massive-scale analytics. This system considers streaming graph data as well as static graph algorithms. Raw, unstructured data is processed in the cloud, and all graph edges are stored in a disk-based system. One or more graph databases running on highly multithreaded servers

produce graph representations of subsets of the data according to application or user-specified requirements. Small portions of the graph database can be downloaded and run locally on a user workstation as memory allows.

Graph algorithms can be run at any level of the hierarchy as long as the required vertices and edges are present. If the data are not resident, and adequate resources exist, portions of the graph must be retrieved from a higher level. We extend our data access time model for this hybrid architecture. We show that for some algorithms, such as connected components, the best choice is to run the algorithm in the graph database, even if the entire graph edge list must be retrieved from disk. In other cases, such as triangle counting, the cost of moving the edge list to the graph database from the SSD-based system is too high. For breadth-first search, there is a clear trade-off. If less than half of the edges are present in the graph database, the computation should be run in the cloud. Otherwise it is faster to update the graph database and run there. Much like an in-memory index to a database table on disk, the multithreaded graph database is faster than the disk-based edge list if a significant portion of the required data is available. The model is parameterized, and different combinations of disks and memory can be studied for different algorithms.

For the first time, we have an understanding of the performance profiles of different storage media. It is clear that spinning hard disk drives cannot compete with in-memory analytics. Solid state drives are possible competitors, but at greater quantities than DRAM. We see that the programming model can have a profound effect on algorithm performance, which can be amplified by hardware design. Future systems will combine massively parallel main memories and SSDs (and possibly phase change memory) to form a highly heterogeneous data processing platform. This shift away from loosely-coupled homogeneous system designs ought to motivate new research into heterogeneous architectures and programming models.

REFERENCES

- [1] D. Bader and K. Madduri, “Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2,” in *Proc. 35th Int’l Conf. on Parallel Processing (ICPP)*, (Columbus, OH), IEEE Computer Society, Aug. 2006.
- [2] J. Ullman and M. Yannakakis, “High-probability parallel transitive closure algorithms,” in *Proceedings of the second annual ACM symposium on Parallel algorithms and architectures*, SPAA ’90, (New York, NY, USA), pp. 200–209, ACM, 1990.
- [3] D. R. Zerbino and E. Birney, “Velvet: Algorithms for de novo short read assembly using de Bruijn graphs,” *Genome Research*, vol. 18, no. 5, pp. 821–829, 2008.
- [4] J. Felsenstein, “Evolutionary trees from DNA sequences: a maximum likelihood approach,” *Journal of molecular evolution*, vol. 17, no. 6, pp. 368–376, 1981.
- [5] D. Sankoff and M. Blanchette, “Multiple genome rearrangement and breakpoint phylogeny,” *J Comput Biol*, vol. 5, no. 3, pp. 555–570, 1998.
- [6] S. Kang, *On the Design of Architecture-Aware Algorithms for Emerging Applications*. PhD thesis, Georgia Institute of Technology, 2011.
- [7] H. Jeong, S. Mason, A.-L. Barabási, and Z. Oltvai, “Lethality and centrality in protein networks,” *Nature*, vol. 411, pp. 41–42, 2001.
- [8] L. Freeman, “A set of measures of centrality based on betweenness,” *Sociometry*, vol. 40, no. 1, pp. 35–41, 1977.
- [9] M. Joy, A. Brock, D. Ingber, and S. Huang, “High-betweenness proteins in the yeast protein interaction network,” *Journal of Biomedicine and Biotechnology*, vol. 2, pp. 96–103, 2005.
- [10] D. Bader and K. Madduri, “A graph-theoretic analysis of the human protein interaction network using multicore parallel algorithms,” in *Proc. 6th Workshop on High Performance Computational Biology (HiCOMB 2007)*, (Long Beach, CA), March 2007.
- [11] R. Guimer, S. Mossa, A. Turtleschi, and L. A. N. Amaral, “The worldwide air transportation network: Anomalous centrality, community structure, and cities’ global roles,” *Proceedings of the National Academy of Sciences*, vol. 102, no. 22, pp. 7794–7799, 2005.

- [12] J. Buhl, J. Gautrais, N. Reeves, R. V. Sol, S. Valverde, P. Kuntz, and G. Theraulaz, “Topological patterns in street networks of self-organized urban settlements,” *The European Physical Journal B - Condensed Matter and Complex Systems*, vol. 49, no. 4, pp. 513–522, 2006.
- [13] K. A. Seaton and L. M. Hackett, “Stations, trains and small-world networks,” *Physica A: Statistical Mechanics and its Applications*, vol. 339, no. 34, pp. 635–644, 2004.
- [14] P. Kaluza, A. Klzsch, M. T. Gastner, and B. Blasius, “The complex network of global cargo ship movements,” *Journal of The Royal Society Interface*, vol. 7, no. 48, pp. 1093–1103, 2010.
- [15] V. Colizza, A. Barrat, M. Barthlemy, and A. Vespignani, “The role of the air-line transportation network in the prediction and predictability of global epidemics,” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 103, no. 7, pp. 2015–2020, 2006.
- [16] J. Shuangshuang, H. Zhenyu, C. Yousu, D. Chavarria-Miranda, J. Feo, and W. Pak Chung, “A novel application of parallel betweenness centrality to power grid contingency analysis,” in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–7, 2010.
- [17] T. Coffman, S. Greenblatt, and S. Marcus, “Graph-based technologies for intelligence analysis,” *Commun. ACM*, vol. 47, pp. 45–47, March 2004.
- [18] M. Faloutsos, P. Faloutsos, and C. Faloutsos, “On power-law relationships of the Internet topology,” in *Proc. ACM SIGCOMM*, (Cambridge, MA), pp. 251–262, ACM, Aug. 1999.
- [19] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, “Graph structure in the web,” *Computer Networks*, vol. 33, no. 1-6, pp. 309–320, 2000.
- [20] R. Albert, H. Jeong, and A.-L. Barabasi, “Error and attack tolerance of complex networks,” *Nature*, vol. 406, no. 6794, pp. 378–382, 2000.
- [21] R. Cohen, K. Erez, D. ben-Avraham, and S. Havlin, “Breakdown of the Internet under intentional attack,” *Phys. Rev. Letters*, vol. 86, no. 16, pp. 3682–3685, 2001.
- [22] Facebook, “User statistics,” October 2012. <http://newsroom.fb.com/content/default.aspx?NewsAreaId=22>.
- [23] D. Ediger, K. Jiang, J. Riedy, D. A. Bader, C. Corley, R. Farber, and W. N. Reynolds, “Massive social network analysis: Mining Twitter for social good,” *Parallel Processing, International Conference on*, pp. 583–593, 2010.

- [24] W. Zachary, “An information flow model for conflict and fission in small groups,” *Journal of Anthropological Research*, vol. 33, pp. 331–338, 1977.
- [25] M. Girvan and M. Newman, “Community structure in social and biological networks,” *Proc. National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.
- [26] A. Clauset, M. E. J. Newman, and C. Moore, “Finding community structure in very large networks,” *Phys. Rev. E*, vol. 70, p. 066111, Dec 2004.
- [27] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, “Parallel community detection for massive graphs,” in *9th International Conference on Parallel Processing and Applied Mathematics (PPAM11)*, Springer, Sept. 2011.
- [28] D. Ediger, K. Jiang, J. Riedy, and D. A. Bader, “Massive streaming data analytics: A case study with clustering coefficients,” in *Workshop on Multithreaded Architectures and Applications (MTAAP)*, (Atlanta, Georgia), Apr. 2010.
- [29] D. Ediger, E. J. Riedy, D. A. Bader, and H. Meyerhenke, “Tracking structure of streaming social networks,” in *5th Workshop on Multithreaded Architectures and Applications (MTAAP)*, May 2011.
- [30] D. A. Bader, J. Berry, A. Amos-Binks, D. Chavarría-Miranda, C. Hastings, K. Madduri, and S. C. Poulos, “STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation,” tech. rep., Georgia Institute of Technology, 2009.
- [31] D. Watts and S. Strogatz, “Collective dynamics of small world networks,” *Nature*, vol. 393, pp. 440–442, 1998.
- [32] P. Konecny, “Introducing the Cray XMT,” in *Proc. Cray User Group meeting (CUG 2007)*, (Seattle, WA), CUG Proceedings, May 2007.
- [33] O. Villa, D. Chavarria-Miranda, and K. Maschhoff, “Input-independent, scalable and fast string matching on the Cray XMT,” in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–12, May 2009.
- [34] J. Mogill and D. Haglin, “Toward parallel document clustering,” in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 1700–1709, May 2011.
- [35] G. Chin, A. Marquez, S. Choudhury, and K. Maschhoff, “Implementing and evaluating multithreaded triad census algorithms on the Cray XMT,” in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–9, May 2009.

- [36] E. Goodman, D. Haglin, C. Scherrer, D. Chavarria-Miranda, J. Mogill, and J. Feo, “Hashing strategies for the Cray XMT,” in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1–8, april 2010.
- [37] K. Jiang, D. Ediger, and D. A. Bader, “Generalizing k -Betweenness centrality using short paths and a parallel multithreaded implementation,” in *The 38th International Conference on Parallel Processing (ICPP 2009)*, (Vienna, Austria), Sept. 2009.
- [38] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarría-Miranda, “A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets,” in *Proc. Workshop on Multithreaded Architectures and Applications (MTAAP’09)*, (Rome, Italy), May 2009.
- [39] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [40] U. Kang, C. Tsourakakis, A. Appel, C. Faloutsos, and J. Leskovec, “Hadi: Fast diameter estimation and mining in massive graphs with Hadoop,” tech. rep., Carnegie Mellon University, 2008.
- [41] J. Cohen, “Graph twiddling in a MapReduce world,” *IEEE Computing in Science and Engineering*, vol. 11, pp. 29–41, 2009.
- [42] Y. Shiloach and U. Vishkin, “An $O(\log n)$ parallel connectivity algorithm,” *J. Algs.*, vol. 3, no. 1, pp. 57–67, 1982.
- [43] F. Chierichetti, R. Kumar, and A. Tomkins, “Max-cover in map-reduce,” 2010.
- [44] W. Bin, D. Yuxiao, K. Qing, and C. Yanan, “A parallel computing model for large-graph mining with MapReduce,” in *Natural Computation (ICNC), 2011 Seventh International Conference on*, vol. 1, pp. 43–47, 2011.
- [45] A. Bialecki, M. Cafarella, D. Cutting, and O. OMalley, “Hadoop: A framework for running applications on large clusters built of commodity hardware,” 2005.
- [46] S. Kang and D. A. Bader, “Large scale complex network analysis using the hybrid combination of a MapReduce cluster and a highly multithreaded system,” in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1–8, 2010.
- [47] V. Batagelj and A. Mrvar, “Pajek – program for large network analysis,” *Connections*, vol. 21, no. 2, pp. 47–57, 1998.
- [48] Analytic Technologies, “UCINET 6 social network analysis software.” <http://www.analytictech.com/ucinet.htm>.

- [49] G. Csardi and T. Nepusz, “The igraph software package for complex network research,” *InterJournal*, vol. Complex Systems, p. 1695, 2006.
- [50] D. Auber, “Tulip,” in *9th Symp. Graph Drawing* (P. Mutzel, M. Jünger, and S. Leipert, eds.), vol. 2265 of *Lecture Notes in Computer Science*, pp. 335–337, Springer-Verlag, 2001.
- [51] P. Shannon and *et al.*, “Cytoscape: a software environment for integrated models of biomolecular interaction networks.,” *Genome Research*, vol. 13, no. 11, pp. 2498–2504, 2003.
- [52] D. Gregor and A. Lumsdaine, “Lifting sequential graph algorithms for distributed-memory parallel computation,” *SIGPLAN Not.*, vol. 40, no. 10, pp. 423–437, 2005.
- [53] J. J. Willcock, T. Hoefer, N. G. Edmonds, and A. Lumsdaine, “Active pebbles: a programming model for highly parallel fine-grained data-driven computations,” in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPOPP ’11, (New York, NY, USA), pp. 305–306, ACM, 2011.
- [54] D. Ediger, K. Jiang, J. Riedy, and D. Bader, “GraphCT: Multithreaded algorithms for massive graph analysis,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2012.
- [55] J. Berry, B. Hendrickson, S. Kahan, and P. Konecny, “Software and algorithms for graph queries on multithreaded architectures,” in *Proc. Workshop on Multithreaded Architectures and Applications*, (Long Beach, CA), March 2007.
- [56] B. Barrett, J. Berry, R. Murphy, and K. Wheeler, “Implementing a portable multi-threaded graph library: The MTGL on Qthreads,” in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–8, May 2009.
- [57] A. Lugowski, D. Alber, A. Buluç, J. R. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis, “A flexible open-source toolbox for scalable complex graph analysis,” in *Proceedings of the Twelfth SIAM International Conference on Data Mining (SDM12)*, pp. 930–941, April 2012.
- [58] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in *Proceedings of the 2010 international conference on Management of data*, SIGMOD ’10, (New York, NY, USA), pp. 135–146, ACM, 2010.
- [59] C. L. Borgman, J. C. Wallis, M. S. Mayernik, and A. Pepe, “Drowning in data: digital library architecture to support scientific use of embedded sensor networks,” in *Proceedings of the 7th ACM/IEEE-CS joint conference on Digital libraries*, JCDL ’07, pp. 269–277, 2007.

- [60] F. Liljeros, C. Edling, L. Amaral, H. Stanley, and Y. Åberg, “The web of human sexual contacts,” *Nature*, vol. 411, pp. 907–908, 2001.
- [61] R. Kouzes, G. Anderson, S. Elbert, I. Gorton, and D. Gracio, “The changing paradigm of data-intensive computing,” *Computer*, vol. 42, pp. 26–34, jan. 2009.
- [62] “GraphCT,” Sept. 2012. ver. 0.8.0.
- [63] U. Kang, C. Tsourakakis, and C. Faloutsos, “Pegasus: A peta-scale graph mining system implementation and observations,” in *Data Mining, 2009. ICDM '09. Ninth IEEE International Conference on*, pp. 229–238, dec. 2009.
- [64] A. A. Hagberg, D. A. Schult, and P. J. Swart, “Exploring network structure, dynamics, and function using NetworkX,” in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, (Pasadena, CA USA), pp. 11–15, Aug. 2008.
- [65] D. Bader and K. Madduri, “SNAP, Small-world Network Analysis and Partitioning: an open-source parallel graph framework for the exploration of large-scale networks,” in *Proc. Int’l Parallel and Distributed Processing Symp. (IPDPS 2008)*, (Miami, FL), Apr. 2008.
- [66] A. Buluç and K. Madduri, “Parallel breadth-first search on distributed memory systems,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11*, (New York, NY, USA), pp. 65:1–65:12, ACM, 2011.
- [67] E. Prud’hommeaux and A. Seaborne, “SPARQL Query Language for RDF,” tech. rep., W3C, 2006.
- [68] P. Boldi and S. Vigna, “The webgraph framework I: compression techniques,” in *Proceedings of the 13th international conference on World Wide Web, WWW '04*, (New York, NY, USA), pp. 595–602, ACM, 2004.
- [69] K. Lang, “Finding good nearly balanced cuts in power law graphs,” tech. rep., Yahoo! Research, 2004.
- [70] J. Feo, D. Harper, S. Kahan, and P. Konecny, “Eldorado,” in *Proceedings of the 2nd Conference on Computing Frontiers, CF '05*, (New York, NY, USA), pp. 28–34, ACM, 2005.
- [71] S. Even, *Graph Algorithms*. New York, NY, USA: W. H. Freeman & Co., 1979.
- [72] D. Bader, S. Kintali, K. Madduri, and M. Mihail, “Approximating betweenness centrality,” in *Proc. 5th Workshop on Algorithms and Models for the Web-Graph (WAW2007)*, vol. 4863 of *lncs*, (San Diego, CA), pp. 134–137, springer, December 2007.
- [73] D. Chakrabarti, Y. Zhan, and C. Faloutsos, “R-MAT: A recursive model for graph mining,” in *Proc. 4th SIAM Intl. Conf. on Data Mining (SDM)*, (Orlando, FL), SIAM, Apr. 2004.

- [74] M. E. J. Newman, “Modularity and community structure in networks,” *Proceedings of the National Academy of Sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.
- [75] B. Bollobas, *Modern Graph Theory*. Springer, 1998.
- [76] S. J. Plimpton and K. D. Devine, “MapReduce in MPI for large-scale graph algorithms,” *Parallel Comput.*, vol. 37, pp. 610–632, Sept. 2011.
- [77] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.
- [78] U. Brandes, “A faster algorithm for betweenness centrality,” *J. Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.
- [79] D. Bader and K. Madduri, “Parallel algorithms for evaluating centrality indices in real-world networks,” in *Proc. 35th Int’l Conf. on Parallel Processing (ICPP)*, (Columbus, OH), IEEE Computer Society, Aug. 2006.
- [80] A.-L. Barabási, “Network databases.” <http://www.nd.edu/networks/resources.htm>, 2007.
- [81] Twitter, “#Goal,” April 2012. <http://blog.uk.twitter.com/2012/04/goal.html>.
- [82] Facebook, “Key facts,” May 2012. <http://newsroom.fb.com/content/default.aspx?NewsAreaId=22>.
- [83] Google, “Introducing the knowledge graph: things, not strings,” May 2012. <http://insidesearch.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html>.
- [84] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
- [85] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, “Summary cache: A scalable wide-area web cache sharing protocol,” in *IEEE/ACM Transactions on Networking*, pp. 254–265, 1998.
- [86] Y. Gu, L. Lu, R. Grossman, and A. Yoo, “Processing massive sized graphs using sector/sphere,” in *Many-Task Computing on Grids and Supercomputers (MTAGS), 2010 IEEE Workshop on*, pp. 1–10, nov. 2010.
- [87] S. Seo, E. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, “Hama: An efficient matrix computation with the MapReduce framework,” in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pp. 721–726, 30 2010-dec. 3 2010.

- [88] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: distributed data-parallel programs from sequential building blocks,” in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys ’07, (New York, NY, USA), pp. 59–72, ACM, 2007.
- [89] M. F. Pace, “BSP vs MapReduce,” *Procedia Computer Science*, vol. 9, no. 0, pp. 246 – 255, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.
- [90] T. Seidl, B. Boden, and S. Fries, “CC-MR — finding connected components in huge graphs with MapReduce,” in *Proceedings of the 2012 European conference on Machine Learning and Knowledge Discovery in Databases - Volume Part I*, ECML PKDD’12, (Berlin, Heidelberg), pp. 458–473, Springer-Verlag, 2012.
- [91] B. Wu, Y. Dong, Q. Ke, and Y. Cai, “A parallel computing model for large-graph mining with MapReduce,” in *Natural Computation (ICNC), 2011 Seventh International Conference on*, vol. 1, pp. 43–47, July 2011.
- [92] S. Suri and S. Vassilvitskii, “Counting triangles and the curse of the last reducer,” in *Proceedings of the 20th international conference on World wide web*, WWW ’11, (New York, NY, USA), pp. 607–614, ACM, 2011.
- [93] A. Ching and C. Kunz, “Apache giraph,” 2012.
- [94] “Graph 500,” 2012. <http://www.graph500.org>.
- [95] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, “Data warehousing and analytics infrastructure at Facebook,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD ’10, (New York, NY, USA), pp. 1013–1020, ACM, 2010.
- [96] S. Choi, P. Cogeze, C. H. Diaz, B. Doering, P. Gargini, M. Graef, B. Huizing, H. Ishiuchi, B. Lin, R. Mahnkopf, J. Roh, J. Shindo, and I. Steff, “International technology roadmap for semiconductors,” tech. rep., Semiconductor Industries Association, 2011.
- [97] R. Koo and S. Toueg, “Checkpointing and rollback-recovery for distributed systems,” *Software Engineering, IEEE Transactions on*, vol. SE-13, pp. 23 – 31, jan. 1987.
- [98] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, “The LAM/MPI checkpoint/restart framework: System-initiated checkpointing,” *International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479–493, Winter 2005.
- [99] J. Hursey, J. Squyres, T. Mattox, and A. Lumsdaine, “The design and implementation of checkpoint/restart process fault tolerance for open mpi,” in

Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pp. 1–8, march 2007.

- [100] B. Wu and Y. Du, “Cloud-based connected component algorithm,” in *Artificial Intelligence and Computational Intelligence (AICI), 2010 International Conference on*, vol. 3, pp. 122–126, October 2010.
- [101] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin, “Crunching large graphs with commodity processors,” in *Proceedings of the 3rd USENIX conference on Hot topic in parallelism, Hot-Par’11*, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2011.
- [102] J. Backus, “Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs,” *Commun. ACM*, vol. 21, pp. 613–641, Aug. 1978.
- [103] G. S. Davidson, K. W. Boyack, R. A. Zacharski, S. C. Helmreich, and J. R. Cowie, “Data-centric computing with the Netezza architecture,” tech. rep., Sandia National Laboratories, 2006.

VITA

David Ediger is a PhD candidate in the School of Electrical and Computer Engineering at the Georgia Institute of Technology in Atlanta, Georgia. His graduate studies are advised by Dr. David A. Bader. He completed a Bachelor of Science degree in Computer Engineering from The George Washington University in 2008, where he graduated *summa cum laude*. There he performed research in high performance reconfigurable computing under the direction of Dr. Tarek El-Ghazawi. He also served as a member of the ECE Department's Undergraduate Curriculum Committee. David completed a Master of Science degree in Electrical and Computer Engineering from Georgia Tech in 2010. He is principal developer of GraphCT, the graph characterization package for highly parallel, massive graph analysis, and co-developer of the STINGER framework for streaming graphs.

Awards and honors include the Georgia Tech President's Fellowship, the GWU Senior Design Award, Outstanding Senior Award, and the Presidential Academic Scholarship. David is a member of the IEEE, Tau Beta Pi, and an Eagle Scout.