A HIGH-PERFORMANCE FRAMEWORK FOR ANALYZING MASSIVE COMPLEX NETWORKS

A Thesis
Presented to
The Academic Faculty

by

Kamesh Madduri

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2008

A HIGH-PERFORMANCE FRAMEWORK FOR ANALYZING MASSIVE COMPLEX NETWORKS

Approved by:

David A. Bader, Advisor
College of Computing
Georgia Institute of Technology

Jonathan Berry
Computer Science and Mathematics Department
Sandia National Laboratories

Richard Fujimoto
College of Computing
Georgia Institute of Technology

Subhash Saini
NASA Advanced Supercomputing Division
NASA Ames Research Center

Richard Vuduc
College of Computing
Georgia Institute of Technology

Date Approved: June 27, 2008

To my parents.

ACKNOWLEDGMENTS

I would like to thank my advisor David Bader for his excellent guidance and support. David first introduced me to research in high performance computing and combinatorics when I spent a summer at the University of New Mexico as an undergraduate NSF REU intern. I have immensely enjoyed working with him, first at UNM and then at Georgia Tech, and David has guided me into working on challenging and meaningful research problems throughout. He deserves most of the credit for my professional development and for directing the successful completion of my dissertation. David is a great role model, and his passion for research, teaching, and academic service continue to inspire me.

I am grateful to Jonathan Berry and Bruce Hendrickson of Sandia National Laboratories for shaping my overall research direction and for advising me on research in graph analysis and multithreaded algorithms. Jon is a great mentor, and the two summers I spent at Sandia labs have been very productive and enjoyable.

I am thankful to NASA Ames Research Center for supporting my research with the NASA graduate research fellowship. I am grateful to Subhash Saini, my NASA mentor, for providing me the opportunity to work at NASA Ames, and for his insightful technical advice.

I am grateful to my proposal and defense committee members – David, Jon, Subhash, Richard Fujimoto, Rich Vuduc, Hongyuan Zha, and Haesun Park – for monitoring my progress, reviewing my work, and reading and commenting on my thesis.

I thank Arlene Washington, Carolyn Young, and Lometa Mitchell for handling all my school-related administrative stuff so efficiently.

I have had the opportunity to collaborate with an amazing bunch of researchers along the way, and have learnt a lot from them. I am especially grateful to John Feo, Guojing Cong, Christine Heitsch, Milena Mihail, K. Subramani, Vipin Sachdeva, Joe Crobak, Virat Agarwal, Varun Kanade, Seunghwa Kang, Shiva Kintali, Viral Shah, and others for working

with me on interesting projects in the past few years.

Graduate school would not have been so much fun without the support of a wonderful group of friends. My room mates over the years – PKT, Dinesh, Ankur, Vipin, Chandu, Avishek and Sriram – deserve a special thank you. I am grateful to members of the High Performance Computing lab, both at UNM and at Georgia Tech, for their support and friendship. A huge thanks to all my undergrad friends, the UNM folks, Virat, Varun, and everybody else at Georgia Tech, for keeping me entertained all the time. Lastly, I must thank Aparna for her companionship; the time we have spent together at Georgia Tech has been delightful and full of pleasant memories, and this year flew by so quickly.

This dissertation is dedicated to my parents, whose love and encouragement gives me immense strength. I love you, and I am fortunate to have such a wonderful family.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Graphs are a fundamental and widely-used abstraction for representing data. We can analytically study interesting aspects of real-world complex systems such as the Internet, social systems, transportation networks, and biological interaction data by modeling them as graphs. Graph-theoretic and combinatorial problems are also pervasive in scientific computing and engineering applications. In this dissertation, we address the problem of analyzing large-scale complex networks that represent interactions between hundreds of thousands to billions of entities. We present SNAP, a new high-performance computational framework for efficiently processing graph-theoretic queries on massive datasets.

Graph analysis is computationally very different from traditional scientific computing, and solving massive graph-theoretic problems on current high performance computing systems is challenging due to several reasons. First, real-world graphs are often characterized by a low diameter and unbalanced degree distributions, and are difficult to partition on parallel systems. Second, parallel algorithms for solving graph-theoretic problems are typically memory intensive, and the memory accesses are fine-grained and highly irregular. The primary contributions of this dissertation are the design and implementation of novel parallel graph algorithms for traversal, shortest paths, and centrality computations, optimized for the small-world network topology, and high-performance multithreaded architectures and multicore servers. SNAP (Small-world Network Analysis and Partitioning) is a modular, open-source framework for the exploratory analysis and partitioning of large-scale networks. With SNAP, we demonstrate the capability to process massive graphs with billions of vertices and edges, and achieve up to two orders of magnitude speedup over state-of-the-art network analysis approaches. We also design a new parallel computing benchmark for characterizing the performance of graph-theoretic problems on high-end systems; study data representations for dynamic graph problems on parallel systems; and apply algorithms in SNAP to solve real-world problems in social network analysis and systems biology.

CHAPTER I

INTRODUCTION

Data-intensive applications have emerged as a prominent computational workload in the petascale computing era. Massive data sets with millions, or even billions, of entities are frequently processed in financial, scientific, security, and several other application areas. Further, data is dynamically generated in many cases, and may be assimilated from multiple sources. Thus, the modeling and analysis of massive, transient data streams raises new and challenging research problems.

There are several analytical methods for the analysis of interaction data. Algorithms in the data stream and related models [143] have been shown to be effective for statistical analysis, and for mining trends in large-scale data sets. Complementarily, a graph or a network representation is a convenient and intuitive abstraction for analyzing data. Unique entities are represented as vertices, and the interactions between them are depicted as edges. The vertices and edges can further be typed, classified, or assigned attributes based on relational information. Analyzing topological characteristics of the network, such as the vertex degree distribution, centrality and community structure, provides valuable insight into the structure and function of the interacting data entities. Common queries on these massive data sets can also be naturally encoded as variants of problems related to graph connectivity, flow, or partitioning. Some examples of graph-theoretic problem formulations include phylogeny reconstruction [142] and analysis of protein interaction networks [179] in computational biology, placement and layout in VLSI chips [127], data mining, and social network analysis [105, 113, 46, 121] applications.

The modeling and analysis of complex interaction data is an active research topic in the social science and statistical physics communities. Real-world systems such as the Internet, socio-economic interactions, and biological networks have been extensively studied from an empirical perspective [4, 145], and this has led to the development of a variety of models to

understand their topological properties and evolution. In particular, technological networks, social interaction graphs, and graph abstractions in biology are shown to exhibit common structural features such as a low graph diameter, skewed vertex degree distribution, self-similarity, and dense subgraphs. Analogous to the small-world (short paths) phenomenon, these real-world data sets are broadly referred to and modeled as *small-world* networks [183, 6]. Practical algorithms for applications such as identification of influential entities, communities, and anomalous patterns in social networks (in general, small-world networks) are well-studied [82, 145].

1.1 Research Challenges in Massive Data Analysis

Graph analysis, and in particular the study of massive graphs, is computationally challenging. In order to effectively utilize a network abstraction for solving massive data stream problems, we need to be able to compactly represent and process large-scale graphs, and also efficiently support fundamental analysis queries on them. On current workstations, it is infeasible to do exact in-core computations on massive graphs (by *large-scale* and *massive*, we refer to graphs where the number of vertices and edges are in the range of 100 million to 10 billion) due to the limited physical memory. In such cases, parallel computing techniques can be applied to obtain exact solutions for memory and compute-intensive graph problems quickly. For instance, recent experimental studies on Breadth-First Search for large-scale sparse graphs show that a parallel in-core implementation [14] is two orders of magnitude faster than an optimized external memory implementation [3].

Parallel graph algorithms is a well-studied research area, and there is extensive literature on work-efficient parallel algorithms for several classical graph problems [107]. However, most of these graph algorithms are designed assuming the parallel random access machine (PRAM) model of computation, which does not realistically model current parallel systems. Thus, very few parallel implementations of PRAM algorithms on current architectures outperform the best sequential algorithm corresponding to the graph problem.

Consider emerging problems arising in the disciplines of social and technological network analysis (e.g., identification of implicit online communities, viral marketing strategies,

quantifying centrality and influence in interaction networks, web algorithms), systems biology (e.g., interactome analysis, epidemiological studies, disease modeling), and homeland security (e.g., detecting trends, anomalous patterns from socio-economic interactions and communication data). In contrast to traditional scientific computing applications, these problems deal with massive amounts of high-dimensional data. The nature of computation is often difficult to characterize - it involves a mix of floating-point, integer and string operations, as well as extensive use of combinatorial algorithms and data structures. Further, there is typically very little work in graph kernels such as connectivity and traversal, with most of the time being spent in fetching data from memory. While parallelism is abundant in graph kernels, it is usually fine-grained in nature and we require architectural support for efficient synchronization to achieve scalable parallel performance. We can clearly observe a mismatch in current parallel architecture designs and the computational characteristics of graph analysis, and hence graph problems achieve only a fraction of the peak performance on current parallel platforms.

Due to their unique structural characteristics, graphs arising from social, biological, and technological systems pose additional computational challenges. In particular, because of the unbalanced degree distribution and a low graph diameter, it is difficult to generate a balanced partitioning of the graph among processors in a parallel system, such that the resulting edge cut is small. Graphs that are representative of high-dimensional data have very little structure that can be exploited, and exhibit relatively low spatial and temporal locality. The memory access patterns are dependent on the graph family, and caching and prefetching on current architectures do not lead to significant performance improvements.

1.2 Overview of Dissertation

With the growing interest in data-intensive applications, there is a compelling need for new algorithms and computational tools that can enable high-performance graph analysis. In this dissertation, we tackle the problem of massive graph analysis with the design and implementation of several high-performance parallel algorithms, optimized for a variety of shared-memory parallel architectures. Our primary contribution is the design of a

new open-source framework SNAP [17, 132] for the exploratory analysis and partitioning of large-scale networks. SNAP incorporates parallel approaches for graph traversal, shortest paths, and centrality computations, optimized for the small-world network topology, and high-performance multithreaded architectures and multicore servers. With SNAP, we demonstrate the capability to process massive graphs with billions of vertices and edges, and achieve up to two orders of magnitude speedup over state-of-the-art network analysis approaches. We also design a new parallel computing benchmark for characterizing the performance of graph-theoretic problems on high-end systems; study data representations for dynamic graph problems on parallel systems; and apply algorithms in SNAP to solve real-world problems in social network analysis and systems biology.

1.2.1 SNAP: Small-world Network Analysis and Partitioning

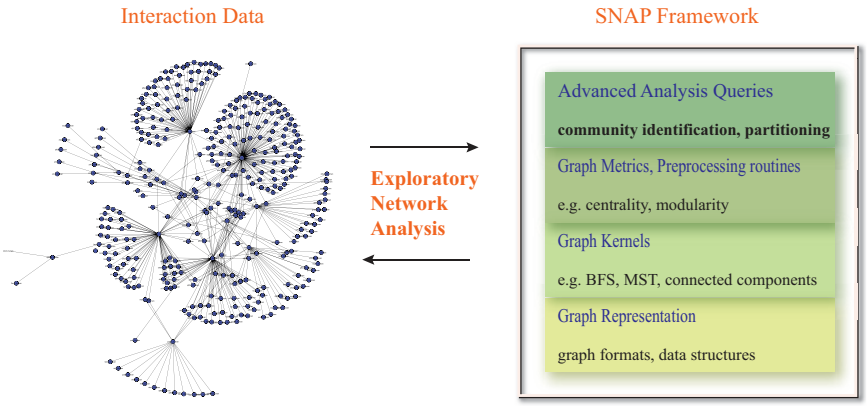


Figure 1: The SNAP graph analysis framework.

We present SNAP (Small-world Network Analysis and Partitioning), an open-source graph framework for exploratory study and partitioning of large-scale networks. SNAP contains parallel implementations of fundamental graph-theoretic kernels and topological analysis metrics (e.g., breadth-first search, connected components, vertex and edge centrality) that are optimized for small-world networks. SNAP is also extensible; the graph kernels are modular, portable across shared memory multicore and symmetric multiprocessor systems, and simplify the design of high-level domain-specific applications. The framework is implemented in C and uses POSIX threads and OpenMP primitives for parallelization. The source code is freely available online [132].

1.2.2 Organization

Figure 1 gives an overview of the SNAP framework. In this dissertation, we discuss the new parallel algorithms underlying the SNAP framework, and focus on graph traversal, shortest paths, centrality, and community identification. Each of these is representative of an abstraction layer in the SNAP framework. The dissertation is organized as follows:

In **Chapter 2**, we present new parallel algorithms and efficient implementations for the Breadth-first Search (BFS), *st*-connectivity, and shortest path problems.

BFS algorithms on massive graphs are characterized by a large memory footprint, irregular memory access patterns, and low spatial and temporal locality. The BFS kernel is representative of a broader class of memory-intensive combinatorial applications, and serves as a valuable benchmark for evaluating the performance of high-end parallel architectures. We conduct an extensive experimental study of parallel breadth-first search algorithms on shared memory architectures such as massively multithreaded systems, multicore, and symmetric multiprocessor systems. We highlight important algorithm changes that are necessitated by the graph topology, and also discuss several architecture-specific optimizations.

We also present an experimental study of the single source shortest path problem with non-negative edge weights (NSSP) on large-scale graphs using the Δ-stepping parallel algorithm. We report performance results on the Cray MTA-2, a multithreaded parallel computer whose architectural features greatly aid the efficient execution of parallel graph algorithms. Our implementation exhibits remarkable parallel speedup when compared with competitive sequential algorithms, for low-diameter sparse graphs. For instance, Δ-stepping on a directed scale-free graph of 100 million vertices and 1 billion edges takes less than ten seconds on 40 processors of the MTA-2, with a relative speedup of close to 30. To our knowledge, these are the first performance results of a shortest path problem on realistic graph instances in the order of billions of vertices and edges.

The crux of exploratory graph analysis is a systematic computational study of the structure and dynamics of a network, using a discriminating selection of topological metrics. In SNAP, we support fast computation of several social network analysis (SNA) metrics, such

as centrality indices, average vertex degree, clustering coefficient, average shortest path length, rich-club coefficient, and assortativity. **Chapter 3** is focused on our contributions in the area of centrality computation and analysis, with particular focus on betweenness centrality. We present new exact and approximate parallel algorithms to compute betweenness centrality on large-scale networks, and demonstrate scalable performance of the centrality approaches on the MTA-2 and the Power 570 systems. We then apply centrality analysis to a real-world dataset, the human protein interaction network (PIN), to better understand the biological aspects of its entities. We report a new topology feature in the yeast and human PINs not found in synthetic scale-free networks: the prevalence of low degree proteins with high-betweenness values. Our results show that existing evolutionary models that produce scale-free networks do not predict the existence of high-betweenness, low-degree vertices found within the yeast and human PINs. The high-betweenness, low centrality vertices also provide some insight into the clustering nature and coreness of the network. We find that vertices with high centrality scores are very likely to be articulation points in the graph, and also have low clustering coefficients.

Community discovery and identification are key analysis routines in understanding the structure of a complex network. There has been a vast amount of research on defining a notion of community in social networks, and on novel algorithms for cluster extraction. An accepted definition of a cluster is a set of vertices with a high local density of edges connecting them, but globally sparse edge connectivity. Clustering coefficients, modularity, and assortativity are some of the metrics proposed to measure the degree of clustering in a network. In **Chapter 4**, to illustrate the capability of the SNAP framework, we detail the design, analysis, and implementation of three novel parallel community identification algorithms. Further, we demonstrate that these parallel approaches are two orders of magnitude faster than competing algorithms – this enables analysis of networks that were previously considered too large to be tractable. We also present more details about the underlying data structures and salient features of the algorithm kernels in SNAP that enable analysis routines such as community identification.

An important contribution of this dissertation is the investigation of new data representations and algorithms for time-evolving networks. We supplement our discussion on parallelization strategies for graph traversal and centrality analysis with new approaches to process massive dynamic networks. In Section 4.5.2, we present parallel approaches to solve the graph traversal and shortest path problems under a series of edge insertions and deletions. We formulate betweenness centrality computation in a dynamic setting in Section 4.5.3, and design outline an approach to efficiently compute it. We also present new graph representations for temporal networks in SNAP in Section 4.5, and evaluate their performance on several parallel systems.

The rest of this chapter is organized as follows. We discuss the topological characteristics of real-world complex networks in Section 1.3, with emphasis on the small-world model and synthetic graph generators. The small-world property, and in particular the low graph diameter, is an important structural feature we exploit in the parallelization of graph traversal and shortest paths. In Section 1.4, we review broad classes of shared-memory high-performance computing architectures. We analyze performance of our parallel algorithms on three HPC systems: the Cray MTA-2, the IBM Power 570, and the Sun Fire T2000. We discuss the distinguishing architectural features of these three systems, and a parallel computational model to analyze algorithms on these systems.

1.3 Modeling real-world networks

Graph abstractions are used to model interactions in a variety of real-world systems such as social networks (friendship circles, organizational networks), the Internet (router topologies, the web-graph, peer-to-peer networks), transportation networks, electrical circuits, genealogical research and computational biology (protein-interaction networks, food webs). These networks seem to be entirely unrelated and indeed represent quite diverse relations, but experimental studies [21, 69, 38, 145, 144] have shown that they share common traits such as a low average distance between the vertices (the *small-world* property), heavy-tailed degree distributions modeled by power laws, and high local densities. Modeling these networks based on experiments and measurements, and the study of interesting phenomena

and observations [40, 48, 153, 187], continue to be active areas of research. Several models [88, 148, 150, 183, 41] have been proposed to generate synthetic graph instances with these characteristics.

Graph generators for creating small-world networks can be broadly classified into the following classes:

Random graph models: A simple and classical model for generating a random graph is the Erdos-Renyi model [68]. We start with a set of n vertices, and for every pair of vertices, we add an edge between them with probability p. This model leads to graphs with a low diameter and a single large component component. It is also easy to analyze algorithm performance on this family of graphs. However, random graphs generated in this manner would have a Poisson degree distribution, whereas real-world networks typically exhibit a power-law distribution. Also, Erdos-Renyi graphs lack community structure, and clustering coefficients in these graphs are not comparable to real-world networks. The Erdos-Renyi random graph model inspired several extensions that match power-law degree distributions observed in real graphs, and also generate graphs with some inherent community structure.

Preferential attachment models: These model the *rich get richer* phenomenon in generation, leading to graphs with skewed degree distributions. Several commonly used models for small-world graphs now use the preferential attachment idea [21]. Informally, the graph generation process can be described as follows: new nodes join the graph at each time step, and preferentially connect to existing nodes with high degree. In addition to a power-law degree distribution, this leads to a network with a low diameter and some inherent resilience – the generated graphs do not break down upon random vertex and edge removals.

Geographical models: These models take the physical topology or some geographical attributes of the entities into account during the generation phase. These are effective for modeling physical networks such as internet router topologies, or power grids.

R-MAT: For our experimental studies, we use the recursive matrix model (R-MAT) [41], a random graph model for generating networks with a power-law degree distribution and community structure. The R-MAT generator creates directed graphs with $n = 2^k$

vertices and m edges, and both the values can be specified as input to the generation algorithm. The generation algorithm is as follows: we start with an empty adjacency matrix and divide it into four equal-sized partitions. One of the four partitions is chosen with probabilities a, b, c, d respectively ($a + b + c + d = 1$). The chosen partition is again sub-divided into four smaller partitions, and the procedure is repeated until we reach a cell in the matrix. An edge is created in the graph corresponding to this cell. This process is repeated m times to generate the complete graph. To smooth out fluctuations in the degree distributions, some noise is added to the a, b, c and d values at each stage of the recursion, followed by renormalization, so that $a + b + c + d = 1$. Intuitively, this would generate *communities* in the network: the partitions a and d represent separate groups of vertices, with b and c being cross-links between these two groups. The recursive nature of the partitions ensures that we automatically generate sub-communities within existing communities. The model can also be extended to generate undirected networks as well as bipartite graphs. Also, given a real-world network, the R-MAT model parameters can be easily computed to fit a power-law degree distribution.

Computationally, we find that the low diameter and the skewed degree distribution are the two important topological properties to consider in the design of new parallel algorithms. We present small-world topology specific optimizations for connectivity and centrality algorithms in Chapters 2 and 3. Also, the presence or absence of community structure impacts the execution time and quality of results obtained using community detection and partitioning algorithms. We discuss these in more detail in Chapter 4.

1.4 High-Performance Computing Systems

As we discussed in the previous section, real-world graphs are typically characterized by a low diameter, heavy-tailed degree distributions modeled by power laws, and self-similarity. They are often very large, with the number of vertices and edges ranging from several hundreds of thousands to billions. On current workstations, it is not possible to do exact in-core computations on these graphs due to the limited physical memory. In such cases, parallel computing techniques can be applied to obtain exact solutions for memory

and compute-intensive graph problems quickly. Solving a problem on a parallel system is typically three orders of magnitude faster than an external memory, out-of-core solution. We review three classes of shared memory systems that are currently popular in landscape of parallel computers. In general, shared memory systems are more supportive than distributed memory platforms for large-scale graph-theoretic computations due to the higher memory bandwidth and lower network latency. Also, a global shared memory abstraction offered by these systems simplifies parallel programming, and we do not need to consider partitioning the graph before the design of parallel algorithms.

1.4.1 Multicore and Symmetric Multiprocessor Servers

For the last few decades, software performance has improved at an exponential rate, primarily driven by the rapid growth in processing power. However, we can no longer rely solely on Moore's law for performance improvements. Fundamental physical limitations such as the size of the transistor and power constraints have now necessitated a radical change in commodity microprocessor architecture to multicore designs. Dual and quad-core processors from Intel [106] and AMD [7] are now ubiquitous in home computing. Also, several novel architectural ideas are being explored for high-end workstations and servers. The Sun UltraSparc T1 [119] with eight processing cores and four threads per core, is a design targeting targeting multithreaded workloads and enterprise applications. The Sony-Toshiba-IBM Cell Broadband Engine [111] is a heterogeneous chip optimized for media and gaming applications. A research proposal from the Intel Tera-scale computing [92] project has eighty cores.

Multicore systems typically have a number of processing cores integrated on to a single chip [106, 7, 23, 119, 111]. Typically, the processing cores have their own private L1 cache and share a common L2 cache [106, 119]. In such a design, the bandwidth between the L2 cache and main memory is shared by all the processing cores. There are primarily three issues that affect performance on multicore systems:

1. *Number of processing cores:* Current systems have two to eight cores integrated on a single chip. Cores typically support features such as simultaneous multithreading

(SMT) or hardware multithreading, which allow for greater parallelism and throughput. In future designs, we may have up to hundred cores on a single chip.

2. *Caching and memory bandwidth:* Memory speeds have been historically increasing at a much slower rate than processor capacity. Memory bandwidth and latency are important performance concerns for several scientific and engineering applications. Caching is known to drastically affect the efficiency of algorithms even on single processor systems [122, 123]. In multicore systems, this will be even more important due to the added bandwidth constraints.

3. *Synchronization:* Implementing algorithms using multiple processing cores will require synchronization between the cores from time to time, which is an expensive operation in shared memory architectures.

In this work, we present results primarily on a Sun multicore server, the Sun Fire T2000 system. This is a homogeneous multicore server that has eight cores running at 1.0 GHz, each of which is four-way multithreaded. There are eight integer units with a six-stage pipeline on-chip, and four threads running on a core share the pipeline. The cores also share a 3 MB L2 cache, and the system has a main memory of 16 GB.

Symmetric multiprocessor (SMP) architectures, in which several processors operate in a true, hardware-based, shared-memory environment are also commonplace in high-performance computing. In comparison to multicore servers, they typically offer a higher memory bandwidth. We execute several experimental studies on a large IBM pSeries SMP system. The IBM Power 570 is a 16-way symmetric multiprocessor with 16 1.9 GHz Power5 cores with simultaneous multithreading (SMT), 32 MB shared L3 cache, and 256 GB shared memory.

1.4.2 Massively Multithreaded Architectures

Massively multithreaded architectures are an alternative platform for solving large-scale graph problems. In comparison to traditional HPC machines and programming models, the Cray MTA-2 (and its successor, the XMT [50]) is a more natural platform for solving

data-intensive applications that are characterized by dynamically varying computations and exhibiting low degrees of spatial locality. The MTA-2 relies on the massive multithreading paradigm of programming to exploit concurrency in applications, and tackles the memory latency issue in a manner that is very different from traditional HPC architectures. It provides the programmer an illusion of a globally addressable flat memory hierarchy, thus avoiding the need for load balanced data partitioning and redistribution among processors. Instead, it tolerates latency through hardware support for multiple outstanding memory requests, facilitated by fine-grained threads and fast context switches on each processor. The MTA-2 supports lightweight word-level synchronization primitives for minimizing memory contention among threads. The massive multithreading approach also supports dynamic load balancing and adaptive parallelism, leading to significant benefits in programmer productivity in the design of algorithms for data-intensive applications. Memory locality and computational intensity of the application have no effect on performance, as long as the programmer identifies and exposes sufficient concurrency at a fine granularity.

The MTA-2 offers two unique features that aid considerably in the design of irregular algorithms: fine-grained parallelism and low-overhead synchronization. The MTA-2 has no data cache; rather than using a memory hierarchy to hide latency, the MTA-2 processors use hardware multithreading to tolerate the latency. The low-overhead synchronization support complements multithreading and makes performance primarily a function of parallelism. Since combinatorial problems often have an abundance of parallelism, these architectural features lead to superior performance and scalability.

Cray's XMT [50, 71] system, formerly called the Eldorado, is a follow-on to the MTA-2 that showcases the massive multithreading paradigm. The XMT is anticipated to scale from 24 to over 8000 processors, providing over one million simultaneous threads and 128 terabytes of globally shared memory. The basic building block of the XMT, the Threadstorm processor, is very similar to the thread-centric MTA processor.

The computational model for the MTA-2 is *thread-centric*, not processor-centric. A thread is a logical entity comprised of a sequence of instructions that are issued in order. An MTA processor consists of 128 hardware *streams* and one instruction pipeline. A stream

is a physical resource (a set of 32 registers, a status word, and space in the instruction cache) that holds the state of one thread. An instruction is three-wide: a memory operation, a fused multiply-add, and a floating point add or control operation. Each stream can have up to 8 outstanding memory operations. Threads from the same or different programs are mapped to the streams by the runtime system. A processor switches among its streams every cycle, executing instructions from non-blocked streams. As long as one stream has a ready instruction, the processor remains fully utilized. No thread is bound to any particular processor. System memory size and the inherent degree of parallelism within the program are the only limits on the number of threads used by a program.

The interconnection network is a partially connected 3-D torus capable of delivering one word per processor per cycle. The system has 4 GBytes of memory per processor. Logical memory addresses are hashed across physical memory to avoid stride-induced hot spots. Each memory word is 68 bits: 64 data bits and 4 tag bits. One tag bit (the full-empty bit) is used to implement synchronous load and store operations. A thread that issues a synchronous load or store remains blocked until the operation completes; but the processor that issued the operation continues to issue instructions from non-blocked streams.

The MTA-2 is closer to a theoretical PRAM machine than a shared memory symmetric multiprocessor system. Since the MTA-2 uses parallelism to tolerate latency, algorithms must often be parallelized at very fine levels to expose sufficient parallelism. However, it is not necessary that all parallelism in the program be expressed such that the system can exploit it; the goal is simply to saturate the processors. The programs that make the most effective use of the MTA-2 are those which express the parallelism of the problem in a way that allows the compiler to best exploit it.

Synchronization is a major limiting factor to scalability in the case of practical shared memory implementations. The software mechanisms commonly available on conventional architectures for achieving synchronization are often inefficient. However, the MTA-2 provides hardware support for fine-grained synchronization through the full-empty bit associated with every memory word. The compiler provides a number of generic routines that operate atomically on scalar variables. We list a few useful constructs that appear in the

algorithm pseudo-codes in subsequent sections.

- The `int_fetch_add` routine (`int_fetch_add`$(\&v, i)$) atomically adds integer i to the value at address v, stores the sum at v, and returns the original value at v (setting the full-empty bit to full). If v is an empty sync or future variable, the operation blocks until v becomes full.

- `readfe`$(\&v)$ returns the value of variable v when v is full and sets v empty. This allows threads waiting for v to become empty to resume execution. If v is empty, the read blocks until v becomes full.

- `writeef`$(\&v, i)$ writes the value i to v when v is empty, and sets v back to full. The thread waits until v is set empty.

- `purge`$(\&v)$ sets the state of the full-empty bit of v to empty.

1.4.3 Analyzing Complexity

To analyze algorithm performance, we use a complexity model similar to the one proposed by Helman and JáJá, [94] which has been shown to provide a good cost model for shared-memory algorithms on current symmetric multiprocessor (SMP) [93, 94, 11, 19] systems. The model uses two parameters: the problem's input size n, and the number p of processors. Running time $T(n,p)$ is measured by the triplet $\langle T_M(n,p)\ ;\ T_C(n,p)\ ;\ B(n,p)\rangle$, where $T_M(n,p)$ is the maximum number of non-contiguous main memory accesses required by any processor, $T_C(n,p)$ is an upper bound on the maximum local computational complexity of any of the processors, and $B(n,p)$ is the number of barrier synchronizations. This model, unlike the idealistic PRAM, is more realistic in that it penalizes algorithms with non-contiguous memory accesses that often result in cache misses, and also considers synchronization events in algorithms. In this dissertation, since our studies are limited to homogeneous multicore processors such as the Sun Niagara system, we make a simplifying assumption and use the Helman-JáJá model for analyzing multicore algorithms as well.

Since the MTA-2 is a shared memory system with no data cache and no local memory, it is comparable to an SMP where all memory reference are remote. Thus, the Helman-JáJá

model can be applied to the MTA-2 with the difference that the magnitudes of $T_M(n,p)$ and $B(n,p)$ are reduced via multithreading. In fact, if sufficient parallelism exists, these costs are reduced to zero and performance is a function of only $T_C(n,p)$. Execution time is then a product of the number of instructions and the cycle time.

The number of threads needed to reduce $T_M(n,p)$ to zero is a function of the memory latency of the machine, about 100 cycles. Usually a thread can issue two or three instructions before it must wait for a previous memory operation to complete; thus, 40 to 80 threads per processor are usually sufficient to reduce $T_M(n,p)$ to zero. The number of threads needed to reduce $B(n,p)$ to zero is a function of intra-thread synchronization. Typically, it is zero and no additional threads are needed; however, hotspots can occur. Usually these can be worked around in software, but they do occasionally impact performance.

1.5 Contributions

The central theme of this dissertation is the design and implementation of SNAP [132, 17], a novel framework for massive complex network analysis. The key contributions of our work are as follows:

- *Novel multithreaded algorithms and efficient implementations of BFS, shortest paths, and* st*-connectivity[14, 134, 51].* Prior studies have predominantly focused on running sequential graph traversal algorithms on graph families that can be easily partitioned, whereas we present new algorithms that process real-world graph instances of differing topologies.

- *Demonstration of the power of massive multithreading for graph algorithms on highly unstructured instances.* On the Cray MTA-2, we achieve impressive parallel performance for BFS and shortest paths on low-diameter random and scale-free graphs. We also evaluate competing parallel approaches on multicore and symmetric multiprocessor architectures, identifying architecture-specific optimizations.

- *Efficient graph traversal on realistic massive graph instances (order of billions of edges).* BFS on a directed, scale-free graph of 400 million vertices and 2 billion edges

takes 5 seconds on 40 processors of the MTA-2, with a relative speedup of approximately 31. Similarly, our Δ-stepping shortest paths implementation on a synthetic directed scale-free graph of 100 million vertices and 1 billion edges takes 9.73 seconds on 40 processors of the MTA-2, with a relative speedup of approximately 30. These are the first results that we are aware of, for solving instances of this scale and also achieving near-linear speedup.

- We present the *first parallel algorithms* for efficiently computing the following centrality metrics: degree, closeness, stress, and betweenness [15]. We optimize the algorithms to exploit typical topological features of real-world graphs. We compute exact betweenness centrality for several large networks such as web crawls, protein-interaction networks, movie-actor and patent citation networks. These graph instances are *three orders of magnitude larger* than the problem sizes that can be processed by current social network analysis packages.

- We present a novel approximation algorithm for *estimating the betweenness centrality* of a given vertex, for both weighted and unweighted graphs [12]. Our approximation algorithm is based on an adaptive sampling technique that significantly reduces the number of single-source shortest path computations for vertices with high centrality. We conduct an extensive experimental study on real-world graph instances, and observe that our random sampling algorithm gives very good betweenness approximations for biological networks, road networks and web crawls.

- We present the case study of betweenness centrality analysis applied to eukaryotic protein-interaction networks (PIN) [16]. Jeong et al. [108] empirically show that betweenness is positively correlated with a protein's essentiality and evolutionary age. We observe that proteins with *high betweenness centrality but low connectivity* are abundant in the human and yeast PINs, and that current small-world network models fail to explain this finding.

- For the problem of community identification in large-scale social networks, we design *three new parallel clustering schemes* [17] (two hierarchical agglomerative approaches,

and one divisive clustering algorithm) that exploit typical topological characteristics of small-world networks. Our novel divisive clustering approach based on approximate edge betweenness centrality is *more than two orders of magnitude* faster than the Newman-Girvan algorithm on the Sun Fire T2000 multicore system, while maintaining comparable clustering quality.

CHAPTER II

GRAPH TRAVERSAL AND SHORTEST PATHS

Breadth-First Search (BFS) [49] is one of the basic paradigms for the design of efficient graph algorithms, and is also representative of a broader class of memory-intensive combinatorial applications. It serves as a valuable benchmark and a representative kernel for evaluating the performance of novel architectures. Recognizing the pervasiveness of graph-theoretic problems in scientific and general-purpose computing, as well as their distinct computational characteristics, Asanovic et al. [9] include graph traversal in their list of *dwarf kernels* (algorithmic methods that capture an important pattern of computation and communication) to evaluate future parallel programming models and architectures. We undertake a comprehensive study of the breadth-first graph traversal problem in this chapter. Using machine-independent algorithmic counts, we evaluate the performance of competing parallel graph traversal algorithms for various graph instances. We then identify topology and parallel architecture-specific optimizations for the various approaches.

We also present an efficient multithreaded implementation of Δ-stepping parallel algorithm [140] for solving the single source shortest path problem on large-scale graph instances. In addition to applications in combinatorial optimization problems, shortest path algorithms are finding increasing relevance in the domain of complex network analysis. Popular graph theoretic analysis metrics such as betweenness centrality [31, 74, 89, 108, 129] are based on shortest path algorithms. Our parallel implementation targets graph families that are representative of real-world, large-scale networks [21, 38, 69, 144, 145]. We also conduct an experimental study of Δ-stepping performance on several graph families, and preliminary results from this work are discussed in [134].

The key contributions in this chapter are as follows:

- *Novel multithreaded algorithms and efficient implementations of BFS, shortest paths, and* st*-connectivity.* Prior studies have predominantly focused on running sequential

graph traversal algorithms on graph families that can be easily partitioned, whereas we consider a variety of real-world graph instances of differing topologies. We also analyze performance using machine independent algorithmic operation counts.

- *Demonstration of the power of massive multithreading for graph algorithms on highly unstructured instances.* On the Cray MTA-2, we achieve impressive parallel performance for BFS and shortest paths on low-diameter random and scale-free graphs. We also evaluate competing parallel approaches on multicore and symmetric multiprocessor architectures, identifying architecture-specific optimizations.

- *Efficient graph traversal on realistic massive graph instances (order of billions of edges).* BFS on a directed, scale-free graph of 400 million vertices and 2 billion edges takes 5 seconds on 40 processors of the MTA-2, with a relative speedup of approximately 31.

- *Solving NSSP for large-scale graph instances in the order of billions of edges.* Δ-stepping on a synthetic directed scale-free graph of 100 million vertices and 1 billion edges takes 9.73 seconds on 40 processors of the MTA-2, with a relative speedup of approximately 30. These are the first results that we are aware of, for solving instances of this scale and also achieving near-linear speedup. Also, the sequential performance of our implementation is comparable to competitive NSSP implementations.

The first half of the chapter discusses parallel algorithms for BFS and *st*-connectivity, while the latter half focuses on the Δ-stepping experimental study.

2.1 Parallel Graph Traversal

2.1.1 Preliminaries

Let $G = (V, E)$ be a graph with n vertices and m edges. Let $s \in V$ denote the *source vertex*, a distinguished vertex in the graph. Each edge $e \in E$ is assigned a weight of unity. A *path* from vertex s to t is defined as a sequence of edges $\langle u_i, u_{i+1} \rangle$, $0 \leq i < l$, where $u_0 = s$ and $u_l = t$. The *length* of a path is the sum of the weights of edges. We use $d(s, t)$ to denote the *distance* between vertices s and t, or the length of the shortest path connecting s and t.

Given the source vertex s, BFS systematically explores the edges of G to discover every vertex that is reachable from s. Optionally, we can compute the distance from s to each reachable vertex, and a *breadth-first tree* rooted at s that contains all the reachable vertices. In a breadth-first traversal, all vertices at a distance k (or *level* k) are first visited, before discovering any vertices at distance $k+1$. The *BFS frontier* is defined as the set of vertices in the current level. BFS is identically defined on both undirected and directed graphs. A first-in first-out (FIFO) queue-based sequential algorithm for BFS takes optimal $O(m+n)$ time in the RAM model of computation.

st-connectivity is a related problem, also applicable to both directed and undirected graphs. Given two vertices s and t, the problem is to determine whether vertex t is reachable from s. Also, we are required to compute $d(s,t)$ if there is a path from s to t. st-connectivity is a basic building block for more complex connectivity and path problems, and has linear worst-case time complexity [22, 80].

The sequential BFS algorithm is a simple linear-time approach that maintains the candidate set of vertices to be explored in a FIFO queue. The queue is initially set to hold the source vertex s. For each vertex v in the queue, its neighbors are inspected and added to the queue if they have not been previously visited. A boolean array of size n is maintained to indicate whether a vertex has been visited or not. The space requirements of sequential BFS are $\mathrm{O}(n)$, and each edge in the graph is visited once for a directed graph (twice for an undirected graph).

The fastest known algorithm for parallel BFS represents the graph as an incidence matrix, and involves repeatedly squaring this matrix, where the element-wise operations are in the min-plus semiring (see [80] for a detailed discussion). This computes the BFS ordering of the vertices in $\mathrm{O}(\log n)$ time in the EREW-PRAM model, but requires $\mathrm{O}\big(n^3\big)$ processors. This makes the algorithm impractical for traversing large-scale graphs.

Prior work on large-scale BFS implementations are either motivated by, or are extensions of, two unique parallel algorithms. In the first approach, vertices are visited level by level as the search progresses, and edges are partitioned (either implicitly or explicitly) among the processors [186, 181]. The edge partitioning can be done in $\mathrm{O}(\log n)$ time with high

probability for load-balanced computation. Thus, the time required to traverse d levels of the graph is $O\left(\frac{m}{p} + d \log p\right)$. However, the problem with this approach is that the running time increases linearly with the number of traversed levels. An alternate algorithm, based on path-limited searches, was proposed by Ullman and Yannakakis [177]. Instead of a level-synchronized search, the graph is explored using multiple path-limited parallel searches, and these searches are finally stitched together to obtain a breadth-first tree from the source vertex. We present shared-memory parallelizations of both these algorithms in this section. In addition, our primary contribution is a work-optimal multithreaded approach for BFS, that is optimized for low-diameter graph topologies and multithreaded architectures. We also demonstrate in our experimental study that it is critical to exploit the graph topology when designing parallel BFS algorithms for massive graphs. Our multithreaded BFS implementation is faster than competing approaches on the Cray MTA-2 and other shared memory systems, and also achieves impressive speedup for several sparse graph families.

2.1.2 Parallel Frontier Expansion (BFS-PF)

Unlike prior parallel approaches to BFS, on the MTA-2 we do not consider load balancing or the use of distributed queues for parallelizing BFS. We employ a simple level-synchronized parallel algorithm (Alg. 1) that exploits concurrency at two key steps in BFS:

1. All vertices at a given *level* in the graph can be processed simultaneously, instead of just picking the vertex at the head of the queue (step 7 in Alg. 1)

2. The adjacencies of each vertex can be inspected in parallel (step 9 in Alg. 1).

We maintain an array d to indicate the level (or distance) of each visited vertex, and process the global queue Q accordingly. Alg. 1 is however a very high-level representation, and hides the fact that thread-safe parallel insertions to the queue and atomic updates of the distance array d are needed to ensure correctness. Alg. 2 details the MTA-2 code required to achieve this (for the critical steps 7 to 12), which is simple and very concise. The loops will not be automatically parallelized as there are dependencies involved. The

Algorithm 1: Level-synchronized parallel BFS based on frontier expansion (BFS-PF).

Input: $G(V, E)$, source vertex s.
Output: Array $d[1..n]$, where $d[v]$ gives the length of the shortest path from s to $v \in V$.

```
1  for all v ∈ V in parallel do
2      d[v] ← ∞;
3      visited[v] ← 0;
4  d[s] ← 0;
5  visited[s] ← 1;
6  Q ← φ;
7  enqueue s ← Q;
8  while Q ≠ φ do
9      for all u ∈ Q in parallel do
10         dequeue u ← Q;
11         for each v adjacent to u in parallel do
12             if atomic_increment(visited[v], 1) = 0 then
13                 d[v] ← d[u] + 1;
14                 enqueue v ← Q;
```

compiler can be forced to parallelize them using the *assert parallel* directive on both the loops. We then note that we have to handle and exploit the nested parallelism in this case. We can explicitly indicate that the iterations of the outer loop can be handled concurrently, and the compiler will dynamically schedule threads for the inner loop. We do this using the compiler directive *loop future* (see Alg. 2) to indicate that the iterations of the outer loop can be concurrently processed.

We use the low-overhead synchronization calls `int_fetch_add`, `readfe()`, and `writeef()` to atomically update the value of d, and insert elements to the queue in parallel. `int_fetch_add` offers synchronized updates to data representing shared counters without using locks. The `readfe` operation atomically reads data from a memory location only after that location's full/empty bit is set full, and sets it back to empty. If the bit is not full to start with, the thread executing the read operation suspends in hardware and is later retried. Similarly, a `writeef` writes to a memory location when the full/empty bit is empty and then sets it to full. A `readfe` should be matched with a `writeef`, or else the program might deadlock.

Algorithm 2: MTA-2 parallel C code for steps 8–14 in Alg. 1.

```
/* While the Queue is not empty */
#pragma mta assert parallel
#pragma mta block dynamic schedule
for (i = startIndex; i < endIndex; i++) {
    u = Q[i];
    /* Inspect all vertices adjacent to u */
    #pragma mta assert parallel
    for (j = 0; j < degree[u]; j++) {
        v = neighbor[u][j];
        /* Check if v has been visited yet? */
        vis = int_fetch_add(&visited[v], 1);
        if (vis == 0) {
            /* Enqueue v */
            Q[int_fetch_add(&count, 1)] = v;
            d[v] = d[u] + 1;
        }
    }
}
```

The MTA compiler automatically collapses the two nested for loops in this case, and schedules the loop iterations in a block-dynamic fashion. Thus the implementation is independent of the vertex degree distribution. We do not need to bother about load balancing in case of graph families with skewed degree distributions, such as real-world scale-free graphs.

2.1.3 Edge-Partitioning (BFS-EP)

We observe that the BFS-PF algorithm will not work well for high-diameter graphs (for instance, consider a chain of vertices with bounded degree). In case of high-diameter graph families, the number of vertices at each BFS level is typically a small number. We do not have sufficient parallelism in the level-synchronized approach to saturate the MTA-2 system. For arbitrary sparse graphs, Ullman and Yannakakis offer high-probability PRAM algorithms for transitive closure and BFS [177] that take $\tilde{O}(n^{\epsilon})$ time with $\tilde{O}(mn^{1-2\epsilon})$ processors, provided $m \geq n^{2-3\epsilon}$. The key idea here is as follows. Instead of starting the search from the source vertex s, we expand the frontier up to a distance d in parallel from a set of randomly chosen *distinguished* vertices (that includes the source vertex s also) in the graph. We then construct a new graph whose vertices are the distinguished vertices, and we have edges between these vertices if they were pair-wise reachable in the previous step.

Now a set of *superdistinguished* vertices are selected among them and the graph is explored to a depth t^2. After this step, the resulting graph would be dense and we can determine the shortest path of the source vertex s to each of the vertices. Using this information, we can determine the shortest paths from s to all vertices.

2.1.4 Parallel *st*-connectivity

Algorithm 3: *st*-connectivity (STCONN-FB): concurrent BFSes from s and t.

Input: $G(V, E)$, vertex pair (s, t)
Output: The smallest number of edges *dist* between s and t, if they are connected

```
1  for all v ∈ V in parallel do
2      color[v] ← WHITE;
3      d[v] ← 0;
4  color[s] ← RED;  color[t] ← GREEN;  Q ← φ;  done ← FALSE;  dist ← ∞;
5  Enqueue s ← Q;  Enqueue t ← Q;
6  while Q ≠ φ and done = FALSE do
7      for all u ∈ Q in parallel do
8          Delete u ← Q;
9          for each v adjacent to u in parallel do
10             color ← readfe(&color[v]);
11             if color = WHITE then
12                 d[v] ← d[u] + 1;
13                 Enqueue v ← Q;
14                 writeef(&color[v], color[u]);
15             else
16                 if color ≠ color[u] then
17                     done ← TRUE;
18                     tmp ← readfe(&dist);
19                     if tmp > d[u] + d[v] + 1 then
20                         writeef(&dist, d[u] + d[v] + 1);
21                     else
22                         writeef(&dist, tmp);
23                 writeef(&color[v], color);
```

We can easily extend the Breadth-First Search algorithm for solving the *st*-connectivity problem too. A naïve implementation would be to start a Breadth-First Search from s, and stop when t is visited. However, we note that we could run BFS concurrently both from s and to t, and if we keep track of the vertices visited and the expanded frontiers on

Algorithm 4: *st*-connectivity (STCONN-MF): Alternate BFSes from s and t.

Input: $G(V, E)$, vertex pair (s, t)
Output: The smallest number of edges $dist$ between s and t, if they are connected

```
1  for all v ∈ V in parallel do
2      color[v] ← WHITE;
3      d[v] ← 0;
4  color[s] ← GRAY;  color[t] ← GRAY;  Qs ← φ;  Qt ← φ;
5  done ← FALSE;  dist ← −1;
6  Enqueue s ← Qs;  Enqueue t ← Qt;  extentS ← 1;  extentT ← 1;
7  while (Qs ≠ φ or Qt ≠ φ) and done = FALSE do
8      Set Q appropriately;
9      for all u ∈ Q in parallel do
10         Delete u ← Q;
11         for each v adjacent to u in parallel do
12             color ← readfe(&color[v]);
13             if color = WHITE then
14                 d[v] ← d[u] + 1;
15                 Enqueue v ← Q;
16                 writeef(&color[v], color[u]);
17             else
18                 if color ≠ color[v] then
19                     dist ← d[u] + d[v] + 1;
20                     done ← TRUE;
21                 writeef(&color[v], color);
22     extentS ← |Qs|;  extentT ← |Qt|;
```

both sides, we can correctly determine the shortest path between s and t. The key steps are outlined in Alg. 3 (termed STCONN-FB), which has both high-level details as well as MTA-specific synchronization constructs. Both s and t are added to the queue initially, and newly discovered vertices are either colored RED (for vertices reachable from s) or GREEN (for vertices that can reach t). When a *back edge* is found in the graph, the algorithm terminates and the shortest path is evaluated. As in the previous case, we encounter nested parallelism here and apply the same optimizations. The pseudo-code is elegant and concise, but must be carefully written to avoid the introduction of race conditions and potential deadlocks.

We also implement an improved algorithm for *st*-connectivity (STCONN-MF, denoting *minimum frontier*, detailed in Alg. 4) that is suited for graphs with irregular degree

distributions. In this case, we maintain two different queues Q_s and Q_t and expand the smaller frontier (Q in Alg. 4 is either Q_s or Q_t, depending on the values of $extentS$ and $extentT$) on each iteration. Thus, STCONN-MF visits fewer vertices and edges compared to STCONN-FB.

2.2 Graph Traversal Experimental Study

2.2.1 Platforms

This section summarizes the experimental setup for our BFS and *st*-connectivity performance results on the Cray MTA-2. We report results on a 40-processor MTA-2, with each processor having a clock speed of 220 MHz and 4GB of RAM.

2.2.2 Problem Instances

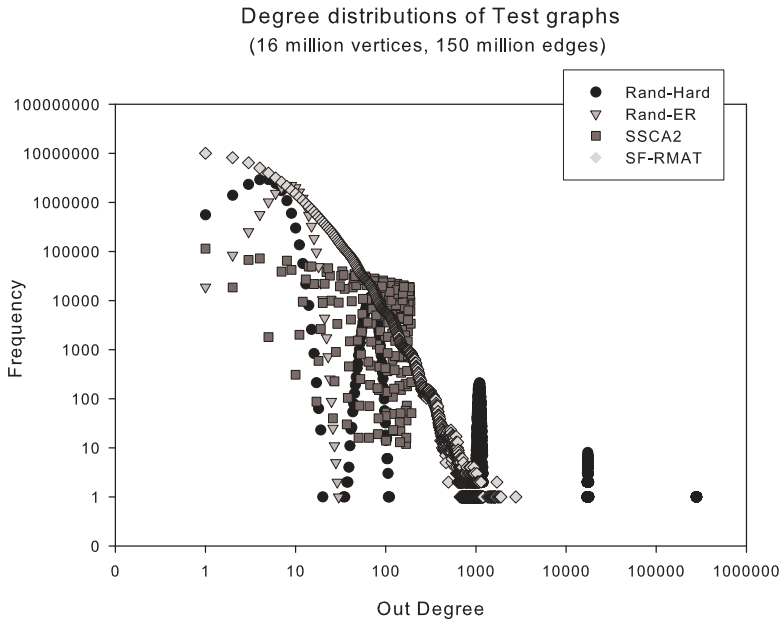


Figure 2: Vertex degree distributions corresponding to graph instances of four different families used in the Breadth-First Search experimental study.

We test our algorithms on four different classes of graphs (see Figure 2):

- Random graphs generated based on the Erdős-Rényi $G(n, p)$ model (Rand-ER): A random graph of m edges is generated with $p = \frac{m}{n^2}$ and has very little structure and locality.

- Scale-free graphs (SF-RMAT), used to model real-world large-scale networks: These graphs are generated using the R-MAT graph model [41]. They have a significant number of vertices of very high degree, although the majority of vertices are low-degree ones. The degree distribution plot on a log-log scale is a straight line with a heavy tail, as seen in Figure 2.

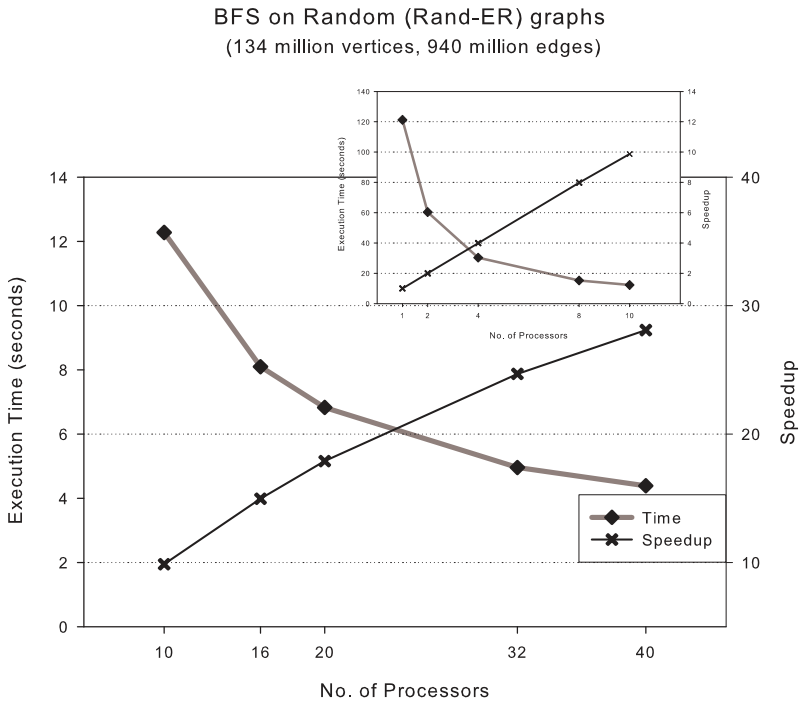
- Synthetic sparse random graphs that are hard cases for parallelization (Rand-Hard): As in scale-free graphs, a considerable percentage of vertices are high-degree ones, but the degree distribution is different.

- DARPA SSCA#2 benchmark (SSCA2) graphs: A typical SSCA#2 graph consists of a large number of highly interconnected clusters of vertices. The clusters are sparsely connected, and these inter-cluster edges are randomly generated. The cluster sizes are uniformly distributed and the maximum cluster size is a user-defined parameter. For the graph used in the performance studies in Figure 2, we assume a maximum cluster size of 10.

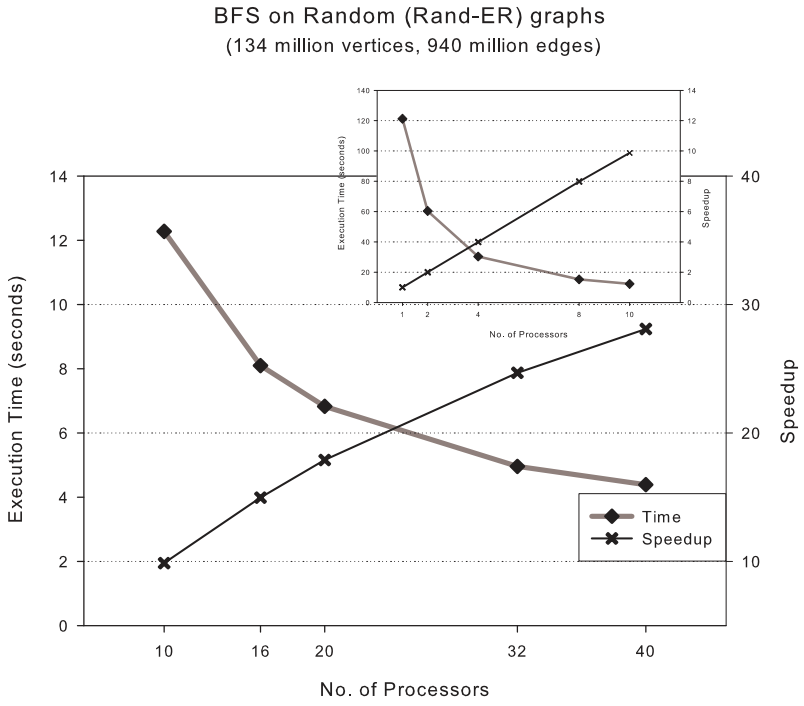
We generate directed graphs in all four cases. Our algorithms work for both directed and undirected graphs, as each vertex stores all its neighbors, and the edges in both directions. In this section, we report results for the undirected case. By making minor changes to our code, we can analyze directed graphs also.

2.2.3 Results and Analysis

Figure 3(a) plots the execution time and speedup attained by the Breadth-First Search algorithm on a random graph of 134 million vertices and 940 million edges (average degree 7). The plot in the inset shows the scaling when the number of processors is varied from 1 to 10, and the main plot for 10 to 40 processors. We define the *Speedup* on p processors of the MTA-2 as the ratio of the execution time on p processors to that on one processor. Since the computation on the MTA is thread-centric, system utilization is also an important metric to study. We observed utilization of close to 97% for single processor runs. We also note that the system utilization was consistently high (around 80% for 40 processor



(a) Random graphs



(b) SF-RMAT graphs

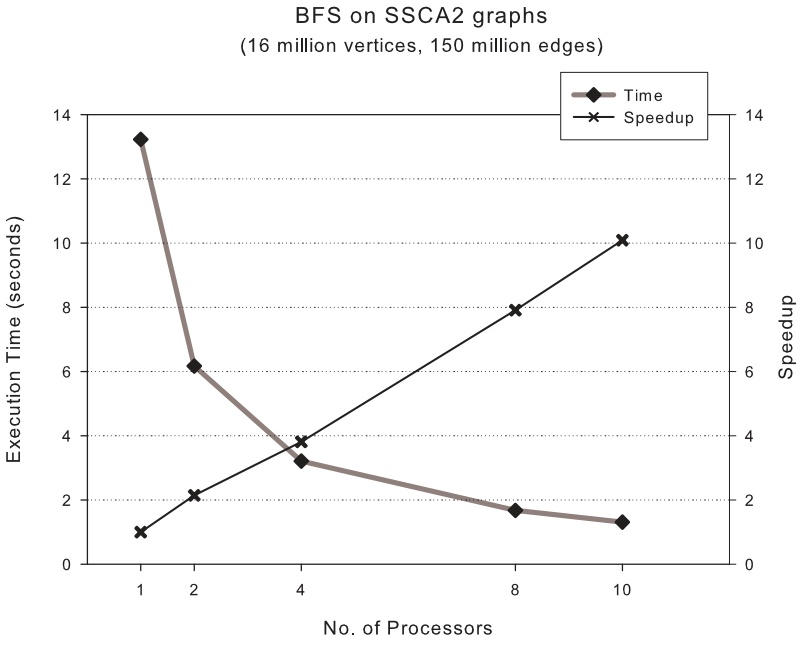
Figure 3: BFS parallel performance: Execution time and speedup on 1-10 processors (inset), and 10-40 processors.

runs) across all runs. We achieve a speedup of nearly 10 on 10 processors for random graphs, 17 on 20 processors, and 28 on 40 processors. This is a significant result, as random graphs have no locality and such instances would offer very limited on no speedup on cache-based SMPs and other shared memory systems. The decrease in efficiency as the number of processors increases to 40 can be attributed to two factors: hot spots in the BFS queue, and a performance penalty due to the use of the future directive for handling nested parallelism.

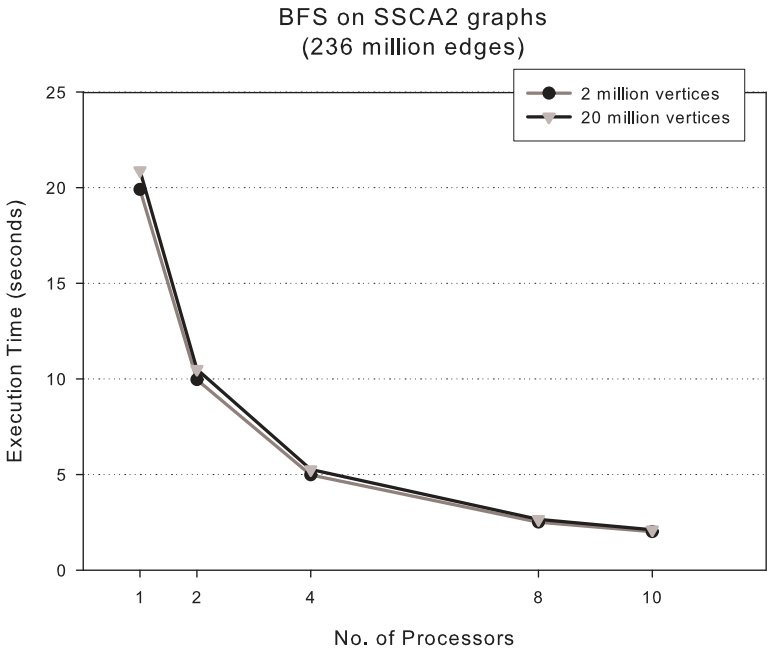
Figure 3(b) gives the BFS execution time for a Scale-free graph of 134 million vertices and 940 million edges, as the number of processors is varied from 1 to 40. The speedups are slightly lower than the previous case, due to the variation in the degree distribution. We have a pre-processing step for high-degree nodes as discussed in the previous sections; this leads to an additional overhead in execution time (when compared to random graphs), as well as insufficient work to saturate the system in some cases. Figure 4 summarizes BFS performance for SSCA#2 graphs. The execution time and speedup (4(a)) are comparable to random graphs. We also varied the user-defined cluster size parameter to see how BFS performs for dense graphs. Figure 4(b) shows that the dense SSCA#2 graphs are also handled well by our BFS algorithm.

Figure 5 shows the performance of BFS as the edge density is varied for Rand-ER and Rand-Hard graphs. We consider a graph of 2.147 billion edges and vary the number of vertices from 16 million to 536 million. In case of Rand-ER graphs, the execution times are comparable as expected, since the dominating term in the computational complexity is the number of edges, 2.147 billion in this case. However, in case of the Rand-Hard graphs, we note an anomaly: the execution time for the graph with 16 million vertices is comparatively more than the other graphs. This is because this graph has a significant number of vertices of very large degree. Even though it scales with the number of processors, since we avoid the use of nested parallelism in this case, the execution times are higher.

Figures 6 and 7 summarize the performance of *st*-connectivity. Note that both the *st*-connectivity algorithms are based on BFS, and if BFS is implemented efficiently, we would expect *st*-connectivity also to perform well. Figure 6 shows the performance of STCONN-MF on random graphs as the number of processors is varied from 1 to 10. Note that

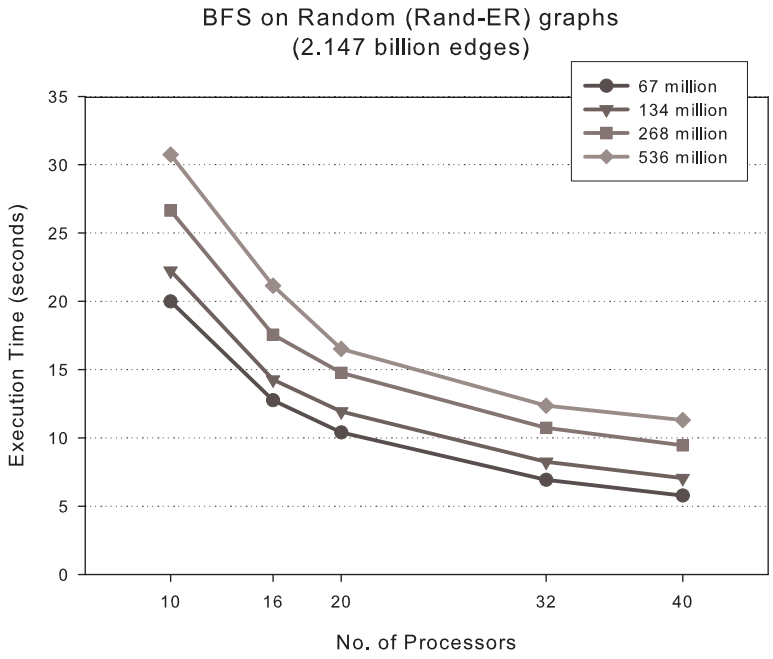


(a) Execution time and speedup

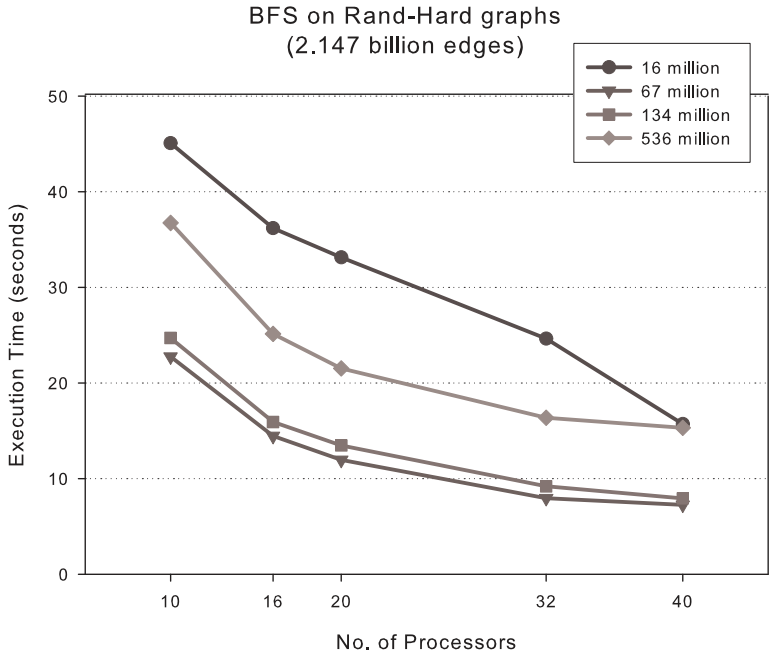


(b) Execution time variation as a function of average degree

Figure 4: BFS parallel performance: SSCA2 graphs.



(a) Rand-ER graphs



(b) Rand-Hard graphs

Figure 5: BFS parallel performance: Execution time variation as a function of average degree.

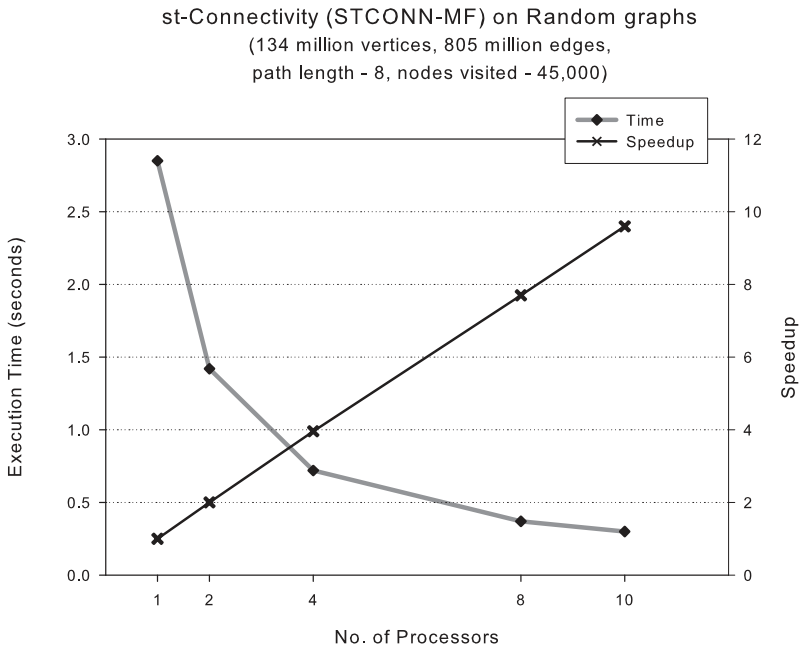


Figure 6: *st*-connectivity performance: Execution time and speedup for Rand-ER graphs.

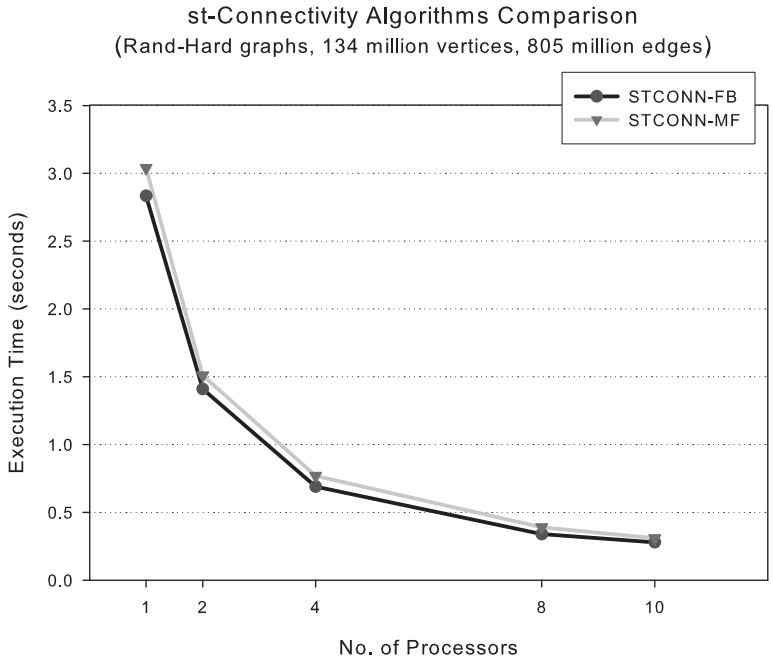


Figure 7: *st*-connectivity performance: Comparison of STCONN-FB and STCONN-MF.

the execution times are highly dependent on (s, t) pair we choose. In this particular case, just 45,000 vertices were visited in a graph of 134 million vertices. The *st*-connectivity algorithm shows near-linear scaling with the number of processors. The actual execution time is bounded by the BFS time, and is dependent on the shortest path length and the degree distribution of the vertices in the graph. In Figure 7, we compare the performance of the two algorithms, concurrent Breadth-First Searches from s and t (STCONN-FB), and expanding the smaller frontier in each iteration (STCONN-MF). Both of them scale linearly with the number of processors for a problem size of 134 million vertices and 805 million edges. STCONN-FB performs slightly better for this graph instance. They were found to perform comparably in other experiments with random and SSCA#2 graphs.

2.3 Parallel Shortest Paths

2.3.1 Preliminaries

Let $G = (V, E)$ be a graph with n vertices and m edges, and let $s \in V$ denote the source vertex. Each edge $e \in E$ is assigned a non-negative real weight by the length function $l : E \rightarrow \mathbb{R}$. Define the *weight of a path* as the sum of the weights of its edges. The single source shortest paths problem with non-negative edge weights computes $\delta(v)$, the weight of the *shortest* (minimum-weighted) path from s to v. $\delta(v) = \infty$ if v is unreachable from s. We set $\delta(s) = 0$.

Most shortest path algorithms maintain a *tentative distance* value for each vertex, which are updated by *edge relaxations*. Let $d(v)$ denote the tentative distance of a vertex v. $d(v)$ is initially set to ∞, and is an upper bound on $\delta(v)$. *Relaxing* an edge $\langle v, w \rangle \in E$ sets $d(w)$ to the minimum of $d(w)$ and $d(v) + l(v, w)$. Based on the manner in which the tentative distance values are updated, most shortest path algorithms can be classified into two types: *label-setting* or *label-correcting*. Label-setting algorithms (for instance, Dijkstra's algorithm) perform relaxations only from *settled* ($d(v) = \delta(v)$) vertices, and compute the shortest path from s to all vertices in exactly m edge relaxations. Based on the values of $d(v)$ and $\delta(v)$, at each iteration of a shortest path algorithm, vertices can be classified

into *unreached* ($d(v) = \infty$), *queued* ($d(v)$ is finite, but v is not settled) or *settled*. Label-correcting algorithms (e.g., Bellman-Ford) relax edges from unsettled vertices also, and may perform more than m relaxations. Also, all vertices remain in a *queued* state until the final step of the algorithm. Δ-stepping belongs to the label-correcting type of shortest path algorithms.

2.3.2 Related Work

Sequential algorithms for the single source shortest path problem with non-negative edge weights are studied extensively, both theoretically [60, 62, 72, 73, 85, 90, 137, 158, 173] and experimentally [43, 61, 77, 83, 84, 189]. Nearly all NSSP algorithms are based on the classical Dijkstra's [62] algorithm. Using Fibonacci heaps [72], Dijkstra's algorithm can be implemented in $O(m + n \log n)$ time. Thorup [173] presents an $O(m + n)$ RAM algorithm for undirected graphs that differs significantly different from Dijkstra's approach. Instead of visiting vertices in the order of increasing distance, it traverses a *component tree*. Meyer [138] and Goldberg [84] propose simple algorithms with linear average time for uniformly distributed edge weights.

Parallel algorithms for solving NSSP are reviewed in detail by Meyer and Sanders [137, 140]. There are no known PRAM algorithms that run in sub-linear time and $O(m+n \log n)$ work. Parallel priority queues [37, 64] for implementing Dijkstra's algorithm have been developed, but these linear work algorithms have a worst-case time bound of $\Omega(n)$, as they only perform edge relaxations in parallel. Several matrix-multiplication based algorithms [76, 91], proposed for the parallel All-Pairs Shortest Paths (APSP), involve running time and efficiency trade-offs. Parallel approximate NSSP algorithms [47, 118, 168] based on the randomized Breadth-First search algorithm of Ullman and Yannakakis [177] run in sub-linear time. However, it is not known how to use the Ullman-Yannakakis randomized approach for exact NSSP computations in sub-linear time.

Meyer and Sanders give the Δ-stepping [140] NSSP algorithm that divides Dijkstra's algorithm into a number of *phases*, each of which can be executed in parallel. For random graphs with uniformly distributed edge weights, this algorithm runs in sub-linear time

with linear average case work. Several theoretical improvements [135, 136, 139] are given for Δ-stepping (for instance, finding shortcut edges, adaptive bucket-splitting), but it is unlikely that they would be faster than the simple Δ-stepping algorithm in practice, as the improvements involve sophisticated data structures that are hard to implement efficiently. On a random d-regular graph instance (2^{19} vertices and $d = 3$), Meyer and Sanders report a speedup of 9.2 on 16 processors of an Intel Paragon machine, for a distributed memory implementation of the simple Δ-stepping algorithm. For the same graph family, we are able to solve problems three orders of magnitude larger with near-linear speedup on the Cray MTA-2. For instance, we achieve a speedup of 14.82 on 16 processors and 29.75 on 40 processors for a random d-regular graph of size 2^{29} vertices and d set to 3.

The literature contains few experimental studies on parallel NSSP algorithms [101, 103, 151, 175]. Prior implementation results on distributed memory machines resorted to graph partitioning [2, 42, 87], and running a sequential NSSP algorithm on the sub-graph. Heuristics are used for load balancing and termination detection [102, 104]. The implementations perform well for certain graph families and problem sizes, but in the worst case, there is no speedup.

In addition to the Δ-stepping algorithm, we recently studied multithreaded implementations of Thorup's algorithm for solving NSSP on undirected graphs and report preliminary results in [51]. Thorup's algorithm constructs and traverses the component hierarchy data structure in order to identify all vertices that can be settled at a given time. This strategy is well suited to a shared-memory environment since the component hierarchy can be constructed only once, then shared by multiple concurrent NSSP computations. On the MTA-2, Δ-Stepping is faster than this implementation for a single source, but Thorup's implementation beats Δ-stepping for simultaneous NSSP runs on 40 processors. We refer the interested reader to [51] for more details on our implementation of Thorup's algoritm.

2.3.3 Review of the Δ-stepping Algorithm

The Δ-stepping algorithm (see Algorithm 5) is an "approximate bucket implementation of Dijkstra's algorithm" [140]. It maintains an array of buckets B such that $B[i]$ stores the

Algorithm 5: Δ-stepping algorithm.

Input: $G(V, E)$, source vertex s, length function $l : E \to \mathbb{R}$
Output: $\delta(v), v \in V$, the weight of the shortest path from s to v

```
1  foreach v ∈ V do
2      heavy(v) ⟵ {⟨v, w⟩ ∈ E : l(v, w) > Δ};
3      light(v) ⟵ {⟨v, w⟩ ∈ E : l(v, w) ≤ Δ};
4      d(v) ⟵ ∞;
5  relax(s, 0);
6  i ⟵ 0;
7  while B is not empty do
8      S ⟵ φ;
9      while B[i] ≠ φ do
10         Req ⟵ {(w, d(v) + l(v, w)) : v ∈ B[i] ∧ ⟨v, w⟩ ∈ light(v)};
11         S ⟵ S ∪ B[i];
12         B[i] ⟵ φ;
13         foreach (v, x) ∈ Req do
14             relax(v, x);
15     Req ⟵ {(w, d(v) + l(v, w)) : v ∈ S ∧ ⟨v, w⟩ ∈ heavy(v)};
16     foreach (v, x) ∈ Req do
17         relax(v, x);
18     i ⟵ i + 1;
19 foreach v ∈ V do
20     δ(v) ⟵ d(v);
```

Algorithm 6: The *relax* routine in the Δ-stepping algorithm.

Input: v, weight request x
Output: Assignment of v to appropriate bucket

```
1 if x < d(v) then
2     B[⌊d(v)/Δ⌋] ← B[⌊d(v)/Δ⌋] \ {v};
3     B[⌊x/Δ⌋] ← B[⌊x/Δ⌋] ∪ {v};
4     d(v) ← x;
```

set of vertices $\{v \in V : v \text{ is queued and } d(v) \in [i\Delta, (i+1)\Delta]\}$. Δ is a positive real number that denotes the "bucket width".

In each *phase* of the algorithm (the inner *while* loop in Algorithm 5, lines 9–14, when bucket $B[i]$ is not empty), all vertices are removed from the current bucket, added to the set S, and *light* edges ($l(e) \leq \Delta$, $e \in E$) adjacent to these vertices are relaxed (see Algorithm 6). This may result in new vertices being added to the current bucket, which are deleted in the next phase. It is also possible that vertices previously deleted from the current bucket may be reinserted, if their tentative distance is improved. *Heavy* edges ($l(e) > \Delta$, $e \in E$) are not relaxed in a phase, as they result in tentative values outside the current bucket. Once the current bucket remains empty after relaxations, all heavy edges out of the vertices in S are relaxed at once (lines 15–17 in Algorithm 5). The algorithm continues until all the buckets are empty.

Observe that edge relaxations in each phase can be done in parallel, as long as individual tentative distance values are updated atomically. The number of phases bounds the parallel running time, and the number of *reinsertions* (insertions of vertices previously deleted) and *rerelaxations* (relaxation of their out-going edges) costs an overhead over Dijkstra's algorithm. The performance of the algorithm also depends on the value of the bucket-width Δ. For $\Delta = \infty$, the algorithm is similar to the Bellman-Ford algorithm. It has a high degree of parallelism, but is inefficient compared to Dijkstra's algorithm. Δ-stepping tries to find a good compromise between the number of parallel phases and the number of re-insertions. Theoretical bounds on the number of phases and re-insertions, and the average case analysis of the parallel algorithm are presented in [140]. We summarize the salient results.

Let d_c denote the maximum shortest path weight, and P_Δ denote the set of paths with weight at most Δ. Define a parameter l_{max}, an upper bound on the maximum number of edges in any path in P_Δ. The following results hold true for any graph family.

- The number of buckets in B is $\lceil d_c/\Delta \rceil$.
- The total number of reinsertions is bounded by $|P_\Delta|$, and the total number of rerelaxations is bounded by $|P_{2\Delta}|$.

- The number of phases is bounded by $\frac{d_c}{\Delta} l_{max}$, i.e., no bucket is expanded more than l_{max} times.

For graph families with random edge weights and a maximum degree of d, Meyer and Sanders [140] theoretically show that $\Delta = \theta(1/d)$ is a good compromise between work efficiency and parallelism. The sequential algorithm performs $O(dn)$ expected work divided between $O(\frac{d_c}{\Delta} \cdot \frac{\log n}{\log \log n})$ phases *with high probability*. In practice, in case of graph families for which d_c is $O(\log n)$ or $O(1)$, the parallel implementation of Δ-stepping yields sufficient parallelism for our parallel system.

2.3.4 Parallel Implementation of Δ-stepping

The bucket array B is the primary data structure used by the parallel Δ-stepping algorithm. We implement individual buckets as *dynamic arrays* that can be resized when needed and iterated over easily. To support constant time insertions and deletions, we maintain two auxiliary arrays of size n: a mapping of the vertex ID to its current bucket, and a mapping from the vertex ID to the position of the vertex in the current bucket (see Figure 8 for an illustration). All new vertices are added to the end of the array, and deletions of vertices are done by setting the corresponding locations in the bucket and the mapping arrays to -1. Note that once bucket i is finally empty after a light edge relaxation phase, there will be no more insertions into the bucket in subsequent phases. Thus, the memory can be reused once we are done relaxing the light edges in the current bucket. Also observe that all the insertions are done in the relax routine, which is called once in each phase, and once for relaxing the heavy edges.

We implement a timed pre-processing step to *semi-sort* the edges based on the value of Δ. All the light edges adjacent to a vertex are identified in parallel and stored in contiguous virtual locations, and so we visit only light edges in a phase. The $O(n)$ work pre-processing step scales well in parallel on the MTA-2.

We also support fast parallel insertions into the request set R. R stores $\langle v, x \rangle$ pairs, where $v \in V$ and x is the requested tentative distance for v. We only add a vertex v to R if it satisfies the condition $x < d(v)$. We do not store duplicates in R. We use a sparse

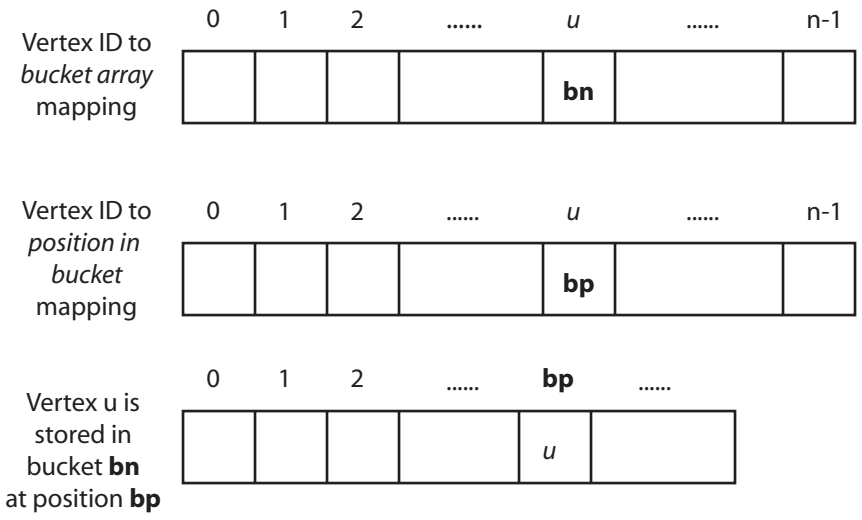


Figure 8: Bucket array and auxiliary data structures in the Δ-stepping algorithm.

set representation similar to one used by Briggs and Torczon [35] for storing vertices in R. This sparse data structure uses two arrays of size n: a *dense* array that contiguously stores the elements of the set, and a *sparse* array that indicates whether the vertex is a member of the set. Thus, it is easy to iterate over the request set, and membership queries and insertions are constant time. Unlike other Dijkstra-based algorithms, we do not relax edges in one step. Instead, we inspect adjacencies (light edges) in each phase, construct a request set of vertices, and then relax *vertices* in the relax step.

All vertices in the request set R are relaxed in parallel in the relax routine. In this step, we first delete a vertex from the old bucket, and then insert it into the new bucket. Instead of performing individual insertions, we first determine the expansion factor of each bucket, expand the buckets, and then add all vertices into their new buckets in one step. Since there are no duplicates in the request set, no synchronization is involved for updating the tentative distance values.

On the MTA-2, accessing the same memory location concurrently by several threads incurs a performance penalty. We call these high-contention memory locations *hot spots*, and need to minimize these to ensure good scalability. For instance, in the relax routine, the bucket size counter may become a hot spot if a significant number of vertices in the current request set are inserted into the same bucket. This is particularly true for low-diameter

graph families such as random and scale-free graphs. However, this leads to a performance penalty only in the case of very large problem instances (random graphs with 500 million to 2 billion edges) using over 30 processors.

To saturate the MTA-2 processors with work and to obtain high system utilization, we need to minimize the number of phases and non-empty buckets, and maximize the request set sizes. Entering and exiting a parallel phase involves a negligible running time overhead in practice. However, if the number of phases is $O(n)$, this overhead dominates the actual running time of the implementation. Also, we enter the relax routine once every phase. There are several implicit barrier synchronizations in the algorithm that are proportional to the number of phases. Our implementation reduces the number of barriers. Our source code for the Δ-stepping implementation, along with the MTA-2 graph generator ports, is freely available online [131].

2.4 Shortest Paths Experimental Study

2.4.1 Platforms

We report parallel performance results on a 40-processor Cray MTA-2 system with 160 GB uniform shared memory. Each processor has a clock speed of 220 MHz and support for 128 hardware threads. The Δ-stepping code is written in C with MTA-2 specific pragmas and directives for parallelization. We compile it using the MTA-2 C compiler (Cray Programming Environment (PE) 2.0.3) with -O3 and -par flags.

The MTA-2 code also compiles and runs on sequential processors without any modifications. Our test platform for the sequential performance results is one processor of a dual-core 3.2 GHz 64-bit Intel Xeon machine with 6GB memory, 1MB cache and running RedHat Enterprise Linux 4 (Linux kernel 2.6.9). We compare the sequential performance of our implementation with the DIMACS reference solver [58]. Both the codes are compiled with the Intel C compiler (icc) Version 9.0, with the flags `-O3`. The source code is freely available online [131].

2.4.2 Problem Instances

We evaluate sequential and parallel performance on several graph families. Some of the generators and graph instances are part of the DIMACS Shortest Path Implementation Challenge benchmark package [57]:

- *Random graphs*: Random graphs are generated by first constructing a Hamiltonian cycle, and then adding $m - n$ edges to the graph at random. The generator may produce parallel edges as well as self-loops. We define the random graph family *Random4-n* such that n is varied, $\frac{m}{n} = 4$, and the edge weights are chosen from a uniform random distribution.

- *Grid graphs*: This synthetic generator produces two-dimensional meshes with grid dimensions x and y. *Long-n* ($x = \frac{n}{16}$, $y = 16$) and *Square-n* grid ($x = y = \sqrt{n}$) families are defined, similar to random graphs.

- *Road graphs*: Road graph families with transit time (*USA-road-t*) and distance (*USA-road-d*) as the length function.

In addition, we also study the following families:

- *Scale-free graphs*: We use the R-MAT graph model [41] for real-world networks to generate scale-free graphs. We define the family *ScaleFree4-n* similar to random graphs.

- *Log-uniform weight distribution*: The above graph generators assume randomly distributed edge weights. We report results for an additional *log-uniform* distribution also. The generated integer edge weights are of the form 2^i, where i is chosen from the uniform random distribution $[1, \log C]$ (C denotes the maximum integer edge weight). We define *Random4logUnif-n* and *ScaleFree4logUnif-n* families for this weight distribution.

2.4.3 Methodology

For sequential runs, we report the execution time of the reference DIMACS NSSP solver (an efficient implementation of Goldberg's algorithm [85], which has expected-case linear

case for some inputs) and the baseline Breadth-First Search (BFS) on every graph family. The BFS running time is a natural lower bound for NSSP codes and is a good indicator of how optimized the shortest path implementations are. It is reasonable to directly compare the execution times of the reference code and our implementation: both use a similar adjacency array representation for the graph, are written in C, and compiled and run in identical experimental settings. Note that our implementation is optimized for the MTA-2 and we make no modifications to the code before running on a sequential machine. The time taken for semi-sorting and mechanisms to reduce memory contention on the MTA-2 both constitute overhead on a sequential processor. Also, our implementation assumes real-weighted edges, and we cannot use fast bitwise operations. By default, we set the value of Δ to $\frac{n}{m}$ for all graph instances. We will show that this choice of Δ may not be optimal for all graph classes and weight distributions.

On the MTA-2, we compare our implementation running time with the execution time of a multithreaded level-synchronized breadth-first search [15], optimized for low-diameter graphs. The multithreaded BFS scales as well as δ-stepping for the core graph families, and the execution time serves as a lower bound for the shortest path running time.

On a sequential processor, we execute the BFS and shortest path codes on all the core graph families, for the recommended problem sizes. However, for parallel runs, we only report results for sufficiently large graph instances in case of the synthetic graph families. We parallelize the synthetic core graph generators and port them to run on the MTA-2.

Our implementations accept both directed and undirected graphs. For all the synthetic graph instances, we report execution times on directed graphs in this paper. The road networks are undirected graphs. We also assume the edge weights to be distributed in $[0, 1]$ in the Δ-stepping implementation. So we have a pre-processing step to scale the integer edge weights in the core problem families to the interval [0, 1], dividing the integer weights by the maximum edge weight.

The first run on the MTA-2 is usually slower than subsequent ones (by about 10% for a typical Δ-stepping run). So we report the average running time for 10 successive runs. We run the code from three randomly chosen source vertices and average the running

time. We found that using three sources consistently gave us execution time results with little variation on both the MTA-2 and the reference sequential platform. We tabulate the sequential and parallel performance metrics in Appendix A, and report execution time in seconds. If the execution time is less than 1 millisecond, we round the time to four decimal digits. If it is less than 100 milliseconds, we round it to three digits. In all other cases, the reported running time is rounded to two decimal digits.

2.4.4 Results and Analysis

2.4.4.1 Sequential Performance

First we present the performance results of our implementation on the reference sequential platform for the core graph families. The BFS, Δ-stepping, and reference DIMACS implementation execution times on the recommended core graph instances are given in Appendix A.1. We observe that the ratio of the Δ-stepping execution time to the Breadth-First Search time varies between 3 and 10 across different problem instances. Also, the DIMACS reference code is about 1.5 to 2 times faster than our implementation for large problem instances in each family. As noted previously, we design an optimized multithreaded implementation of the shortest path algorithm, and some of the mechanisms specific to the MTA-2 may be an overhead on the reference sequential platform. Thus, the sequential execution times quantify the additional work due to parallelization.

Table 8 summarizes the performance for random graph instances. For the Random4-n family, n is varied from 2^{11} to 2^{21}, the maximum edge weight is set to n, and the graph density is constant. For the largest instance, Δ-stepping execution time is 1.7 times slower than the reference implementation and 5.4 times the BFS execution time. For the Random4-C family, we normalize the weights to the maximum integer weight. We do not observe any trend similar to the reference implementation, where the execution time gradually rises as the maximum weight increases. This suggests that the Δ-stepping algorithm performance is independent of maximum integer edge weight, provided the edge weights follow a uniform random distribution and Δ is set appropriately.

The sequential performance of Δ-stepping on Long grid graphs (Table 9) is similar to

that on Random graphs. However, the reference implementation is slightly faster on long grids. For square grids and road networks, the Δ-stepping to BFS ratio is comparatively higher (e.g., BFS to Δ-stepping ratio is 4.71 for the largest Square-n graph, and 3.74 for the largest Random4-n graph) than the Random and Long grid families.

Figure 9 and Figure 10 summarize the key observations from the tables in Appendix A.1. Comparing execution time across graphs of the same size in Figure 9, we find that the Δ-stepping running time for the Random4-n graph instance is slightly higher than the rest of the families. The Δ-stepping running time is also comparable to the execution time of the reference implementation for all graph families. Figure 10 plots the execution time normalized to the problem size (or the running time per edge) for Random4-n and Long-n families. Observe that the Δ-stepping implementation execution time scales with problem size at a faster rate compared to BFS or the DIMACS reference implementation. This suggests a slight increase in additional computation as the problem size is scaled up.

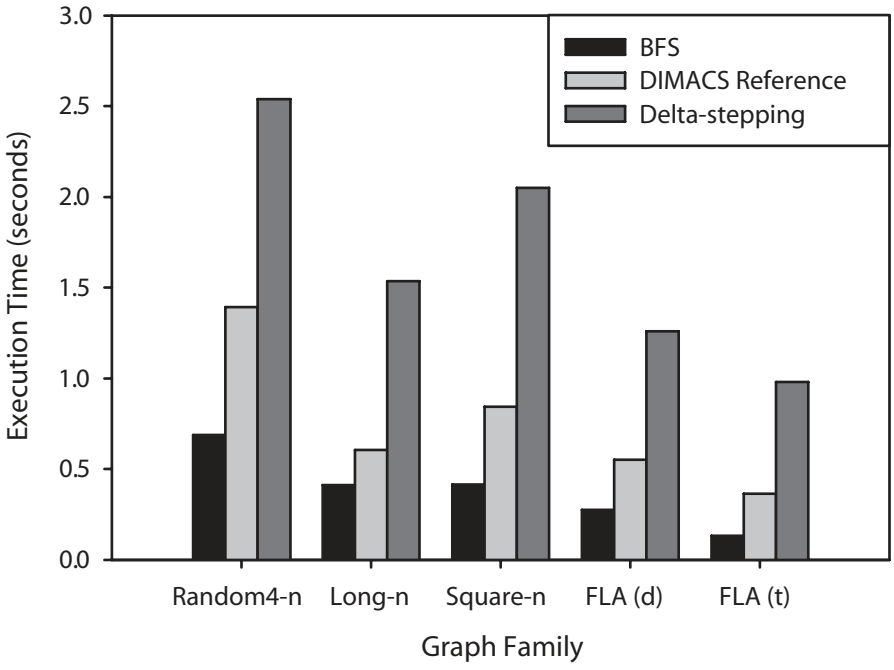
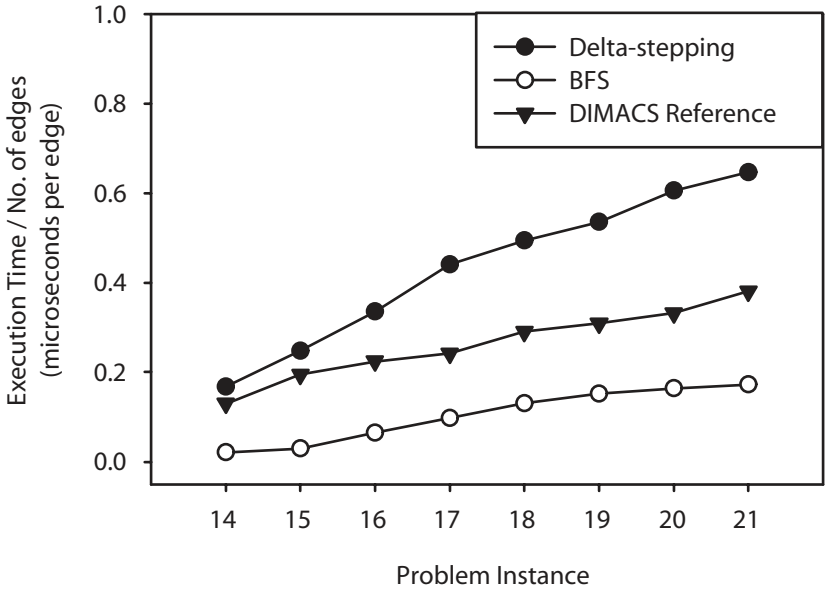
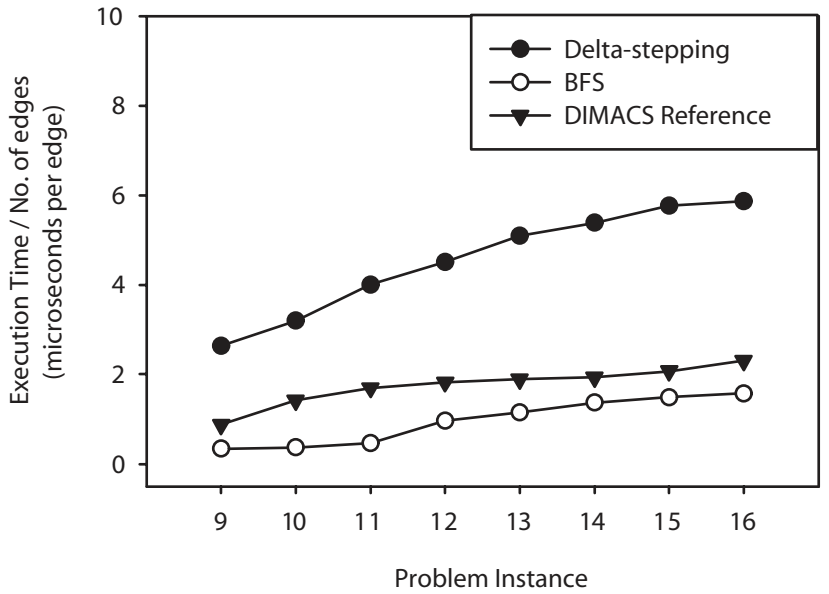


Figure 9: Sequential performance of our Δ-stepping implementation, on the core graph families. All the synthetic graphs are directed, with 2^{20} vertices and $\frac{m}{n} \approx 4$. FLA(d) and FLA(t) are road networks corresponding to Florida, with 1070376 vertices and 2712768 edges.



(a) Random4-n family. Problem instance i denotes a directed graph of 2^i vertices, $m = 4n$ edges, and maximum weight $C = n$.



(b) Long-n family. Problem instance i denotes a grid with $x = 2^i$ and $y = 16$. $n = xy$ and $\frac{m}{n} \approx 4$.

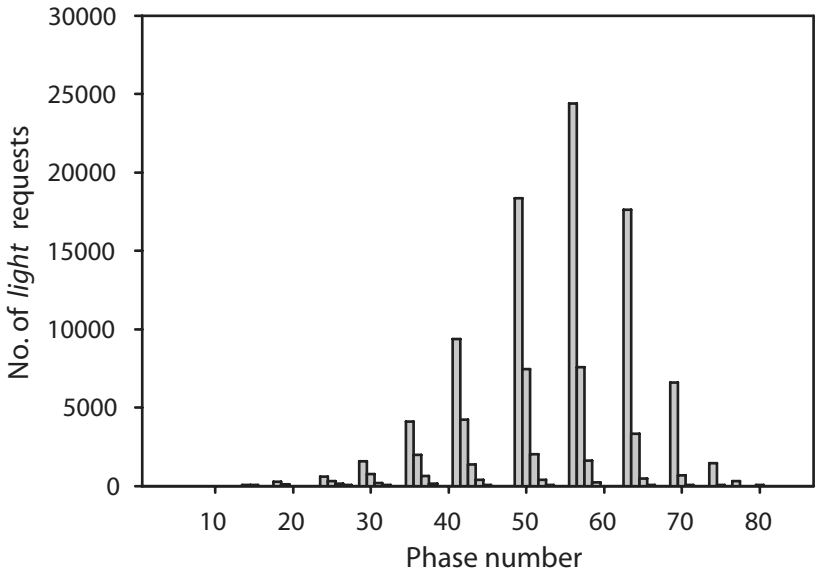
Figure 10: Sequential execution time of our Δ-stepping implementation as a function of problem size.

2.4.4.2 Δ-stepping Analysis

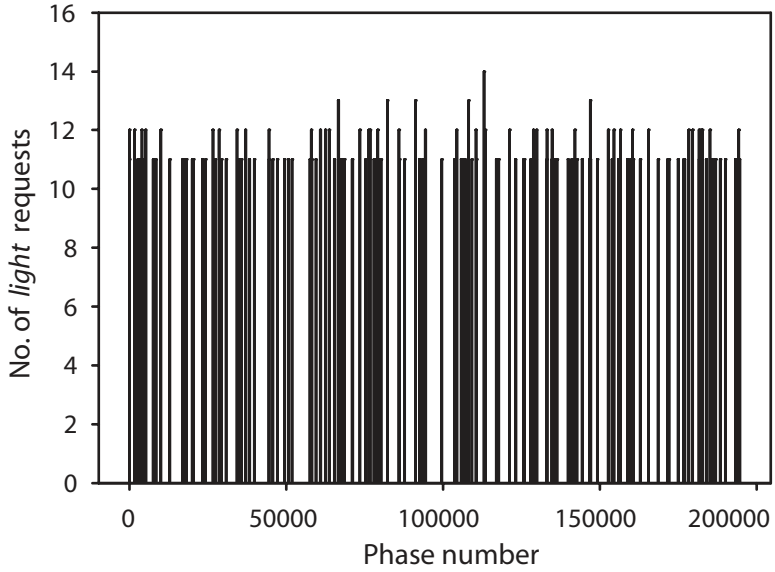
To better understand the algorithm performance across graph families, we study machine-independent algorithm operation counts. The parallel performance is dependent on the value of Δ, the number of phases, the size of the request set in each phase.

Size of request sets. Figure 11 and Figure 12 plot the size of the light request set in each phase, for each core graph family. The choice of Δ in these experiments is motivated by the observation of Meyer and Sanders [140] that for graph families with random edge weights and a maximum degree of d, $\Delta = \theta(1/d)$ would be a good compromise between work efficiency and parallelism. Since $d \approx m/n (\approx 4)$ for most of the test graph instances, Δ is set to 0.25 by default for all runs. We also evaluate the performance of the algorithm as the value of Δ is varied (see Figure 2.4.4.2 and Figure 2.4.4.2). If the request set size is less than 10, it is not plotted.

Consider the random graph family (Figure 11(a)). It executes in 84 phases, and the request set sizes vary from 0 to 27,000. Observe the recurring pattern of nearly three bars stacked together in the plot. This indicates that all the light edges in a bucket are relaxed in roughly three phases, and the bucket then becomes empty. The size of the relax set is relatively high for several phases, which provides scope for exploiting multithreaded parallelism. The relax set size plot of a similar problem instance from the Long grid family (Figure 11(b)) stands in stark contrast to the random graph plot. It takes about 200,000 phases to execute (compared to the 84 phases for the random graph), and the maximum request size is only 15. Both of these values indicate that our implementation performance is significantly dependent on the graph diameter, and that the parallel performance would be poor on long grid graphs (e.g. meshes with a very high aspect ratio). On square grids (Figure 12(a)), Δ-stepping takes fewer phases, and the request set sizes go up to 500. For a road network instance (NE USA-road-d, Figure 12(b)), the algorithm takes 23,000 phases to execute, and only a few phases (about 30) have request set counts greater than 1000. As expected, the number of phases are proportional to the graph diameter in all the cases.

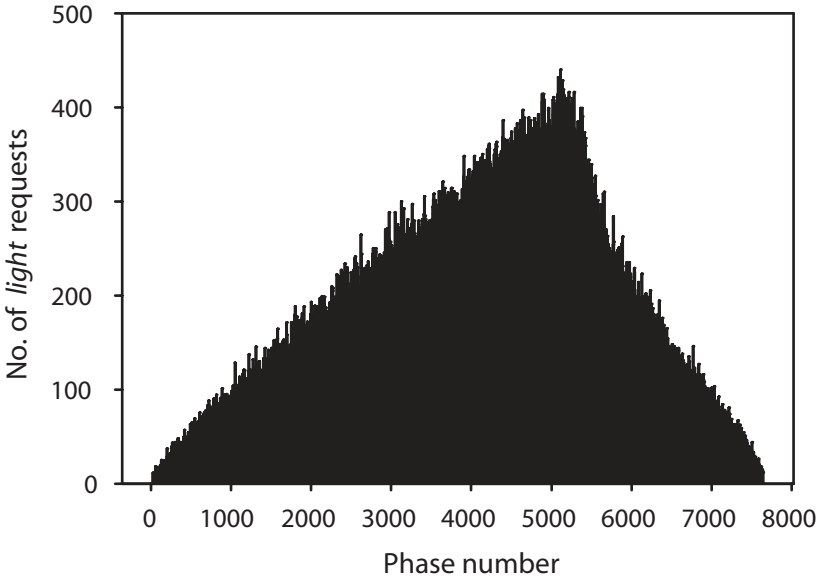


(a) Random4-n family, $n = 2^{20}$.

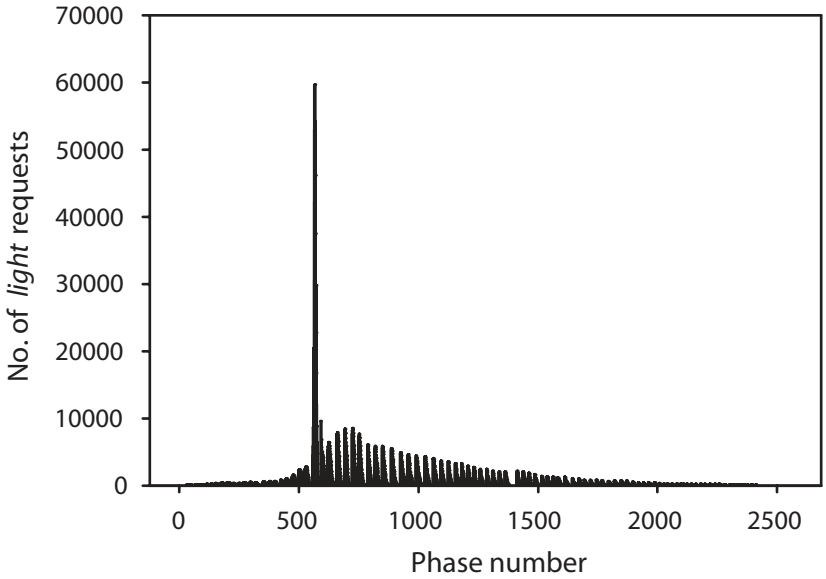


(b) Long-n family, $n = 2^{20}$.

Figure 11: Δ-stepping algorithm: Size of the light request set at the end of each phase, for the core graph families. Request set sizes less than 10 are not plotted.



(a) Square-n family, $n = 2^{20}$.

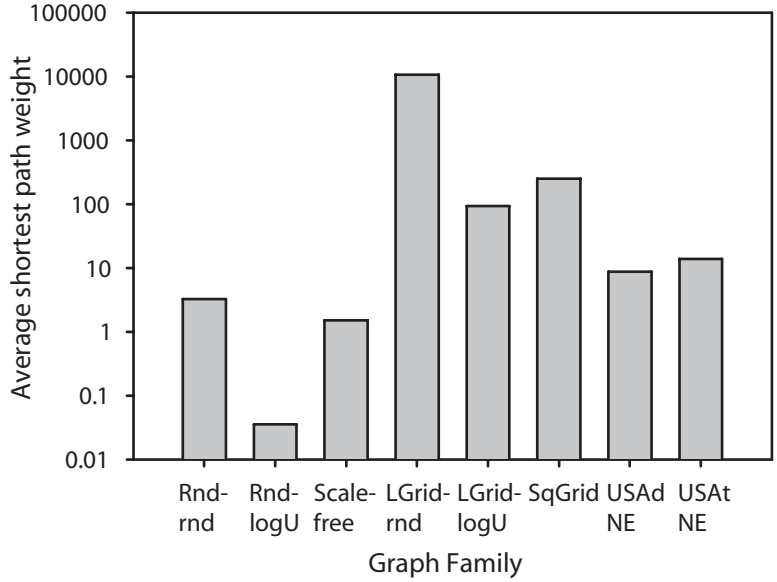


(b) USA-road-d family, Northeast USA (NE). $n = 1524452$, $m = 3897634$.

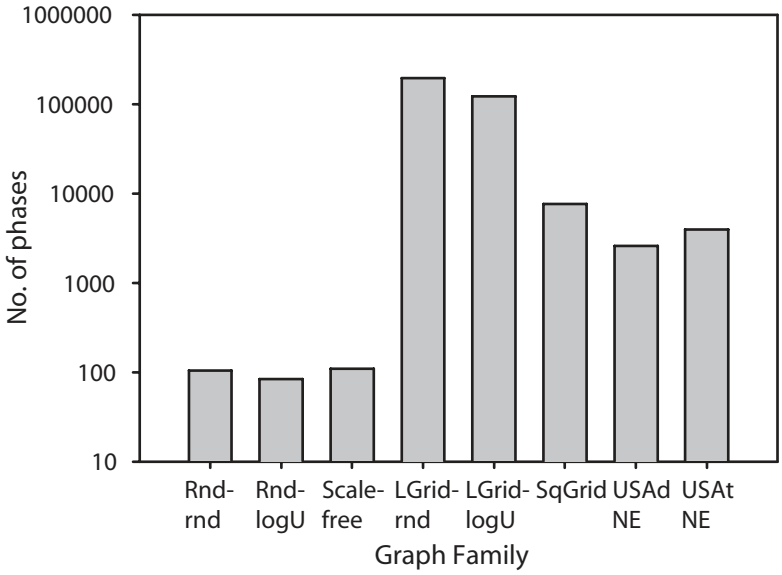
Figure 12: Δ-stepping algorithm: Size of the light request set at the end of each phase, for the core graph families. Request set sizes less than 10 are not plotted.

Algorithm operation counts. Figure 13 and Figure 14 plot several key Δ-stepping operation counts for various graph classes. Along with the core graph families, we include ScaleFree4-n, RandomlogUnif4-n, and LonglogUnif4-n graph classes. All synthetic graphs are roughly of the same size. Figure 13(a) plots the average shortest path weight for various graph classes. Scale-free and Long grid graphs are on the two extremes, with the graph diameter again being the determining factor. A log-uniform edge weight distribution also results in low average edge weight. The number of phases (see Figure 13(b)) is highest for Long grid graphs. The number of buckets shows a similar trend as the average shortest path weight. Figure 14(b) plots the total number of insertions for each graph family. The number of vertices is 2^{20} for all graph families (slightly higher for the road network), and so Δ-stepping results in roughly 20% overhead in insertions for all the graph families with random edge weights. Note the number of insertions for graphs with log-uniform weight distributions. Δ-stepping performs a lot of excess work for these families, because the value of Δ is quite high for this particular distribution.

Influence of Δ. We next evaluate the performance of the algorithm as Δ is varied (tables in Appendix A.2). Figure 2.4.4.2 and Figure 2.4.4.2 plot the execution time of various graph instances on a sequential machine, and one processor of the MTA-2. Δ is varied from 0.1 to 10 in each case. We find that the absolute running times on a 3.2 GHz Xeon processor and the MTA-2 are comparable for random, square grid and road network instances. However, on long grid graphs (Figure 15(b)), the MTA-2 execution time is two orders of magnitude greater than the sequential time. The number of phases and the total number of relaxations vary as Δ is varied (Tables 13, 12, and 14). On the MTA-2, the running time is not only dependent on the work done, but also on the number of phases and the average number of relax requests in a phase. For instance, in the case of long grids (see Figure 15(b), with execution time plotted on a log scale), the running time decreases significantly as the value of Δ is decreased, as the number of phases reduce. On a sequential processor, however, the running time is only dependent on the work done (number of insertions). If the value of Δ is greater than the average shortest path weight, we perform excess work and the running

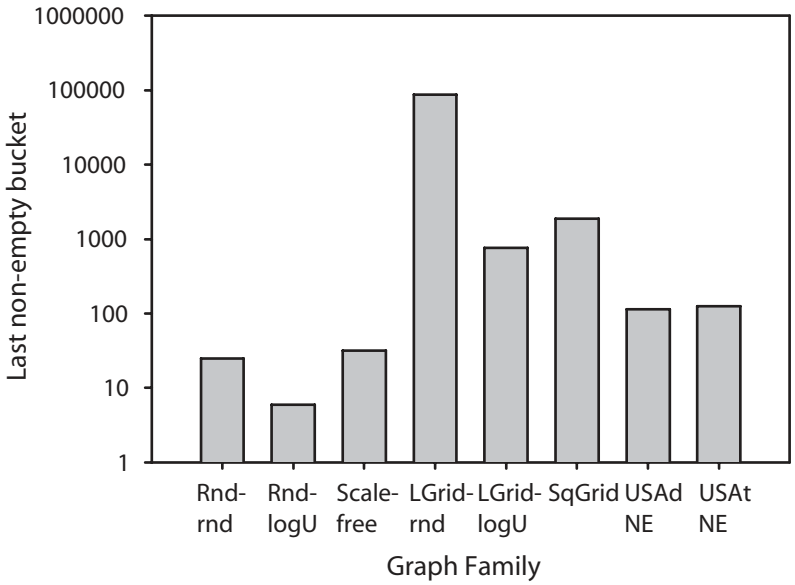


(a) Average shortest path weight ($\frac{1}{n} * \sum_{v \in V} \delta(v)$).

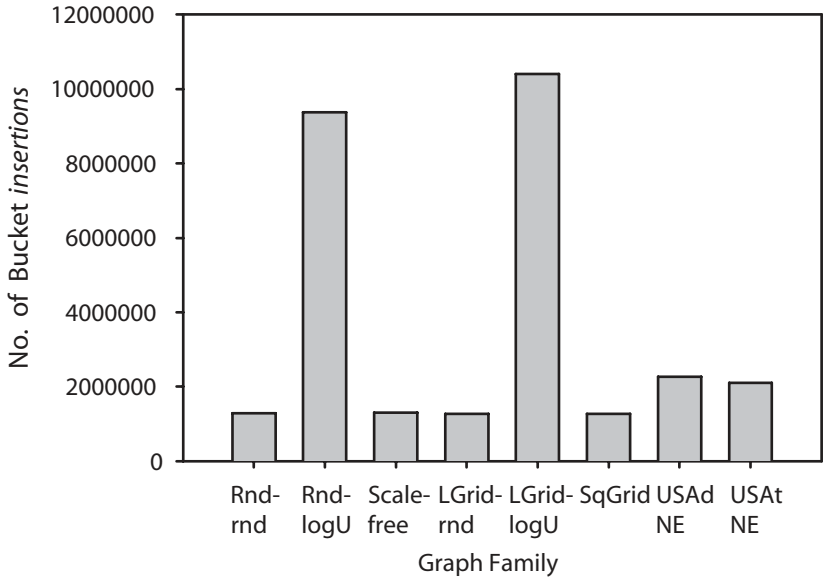


(b) No. of phases.

Figure 13: Δ-stepping algorithm performance statistics for various graph families. All synthetic graph instances have n set to 2^{20} and $m \approx 4n$. Rnd-rnd: Random graph with random edge weights, Rnd-logU: Random graph with log-uniform edge weights, Scale-free: Scale-free graph with random edge weights, LGrid: Long grid, SqGrid: Square grid, USA NE: 1524452 vertices, 3897634 edges.



(a) Last non-empty bucket.



(b) Number of relax requests.

Figure 14: Δ-stepping algorithm performance statistics for various graph classes. All synthetic graph instances have n set to 2^{20} and $m \approx 4n$. Rnd-rnd: Random graph with random edge weights, Rnd-logU: Random graph with log-uniform edge weights, Scale-free: Scale-free graph with random edge weights, LGrid: Long grid, SqGrid: Square grid, USA NE: 1524452 vertices, 3897634 edges. Plot (b) uses a linear scale.

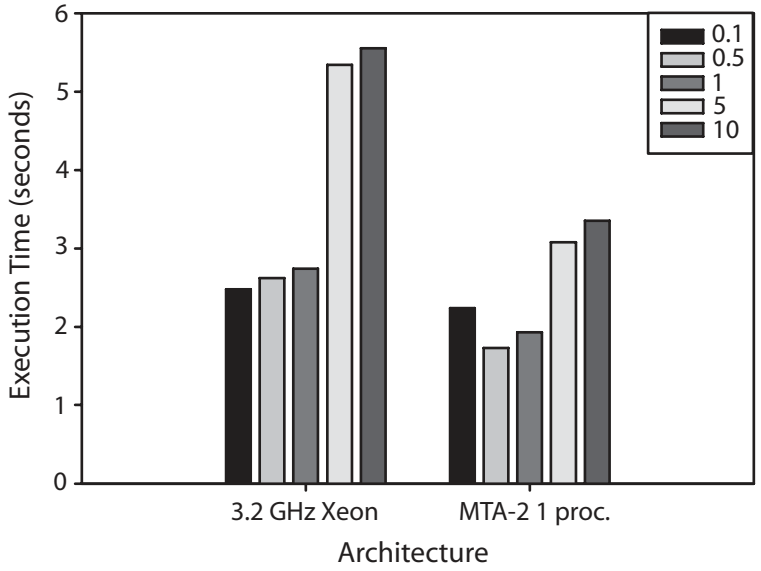
time noticeably increases (observe the execution time for $\Delta = 5, 10$ on the random graph and the road network). The optimal value of Δ (and the execution time on the MTA-2) is also dependent on the number of processors. For a particular Δ, it may be possible to saturate a single processor of the MTA-2 with the right balance of work and phases. The execution time on a 40-processor run may not be minimal with this value of Δ.

2.4.4.3 Parallel Performance

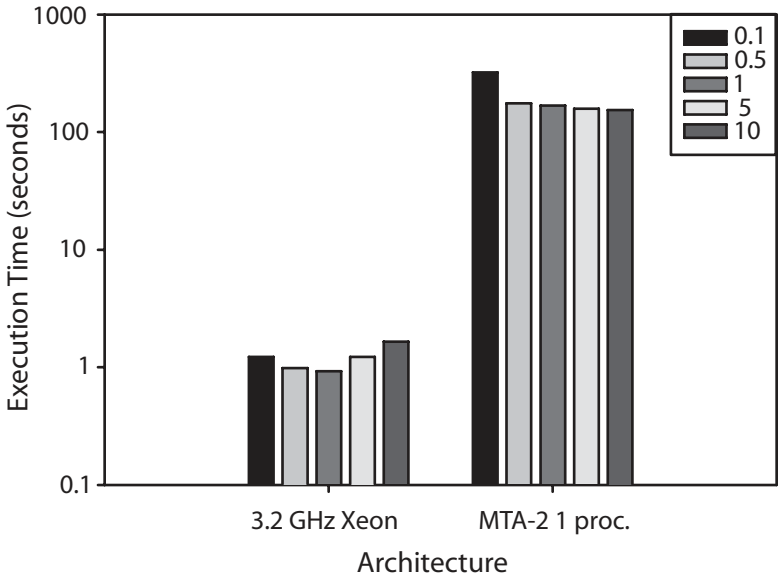
In this section, we discuss the parallel scaling of Δ-stepping in detail (see tables in Appendix A.3). We ran Δ-stepping and the level-synchronous parallel BFS on graph instances from the core families, scale-free graphs, and graphs with log-uniform edge weight distributions. Define the speedup on p processors of the MTA-2 as the ratio of the execution time on 1 processor to the execution time on p processors. Since the computation on the MTA-2 is thread-centric rather than processor-centric, note that the single processor run is also parallel. In all graph classes except long grids, there is sufficient parallelism to saturate a single processor of the MTA-2 for reasonably large problem instances.

Unstructured Instances

As expected from the discussion in the previous section, Δ-stepping performs best for low-diameter random and scale-free graphs with randomly distributed edge weights (see Figure 17 and Figure 18). We attain a speedup of approximately 31 on 40 processors for a directed random graph of nearly a billion edges, and the ratio of the BFS and Δ-stepping execution time is a constant factor (about 3-5) throughout. The implementation performs equally well for scale-free graphs, that are more difficult to handle due to the irregular degree distribution. The execution time on 40 processors of the MTA-2 for the scale-free graph instance is only 1 second slower than the running time for a random graph and the speedup is approximately 30 on 40 processors. We have already shown that the execution time for smaller graph instances on a sequential machine is comparable to the DIMACS reference implementation, a competitive NSSP algorithm. Thus, attaining a speedup of 30 for a realistic scale-free graph instance of one billion edges (Figure 18) is a remarkable result.

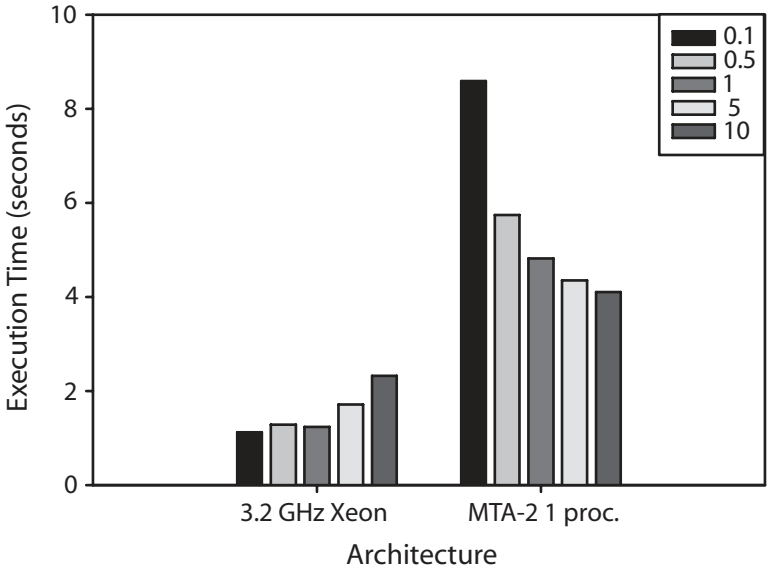


(a) Random4-n family. 2^{20} vertices.

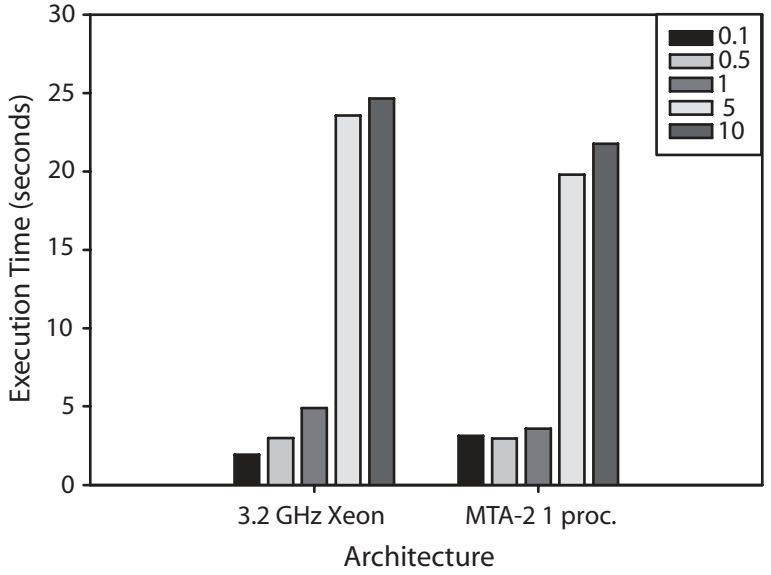


(b) Long-n family. 2^{20} vertices.

Figure 15: A comparison of the execution time on the reference sequential platform and a single MTA-2 processor, as the bucket-width Δ is varied.

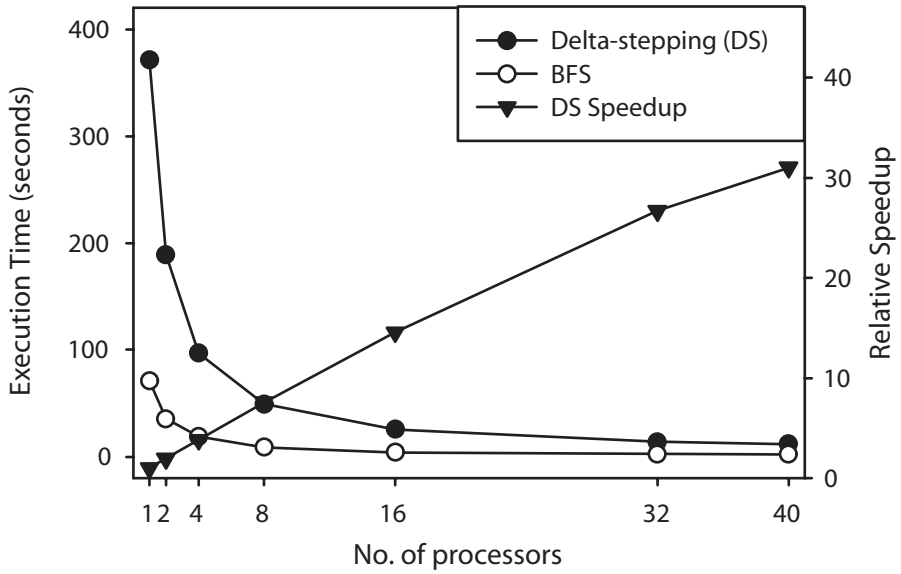


(a) Square-n family. 2^{20} vertices.

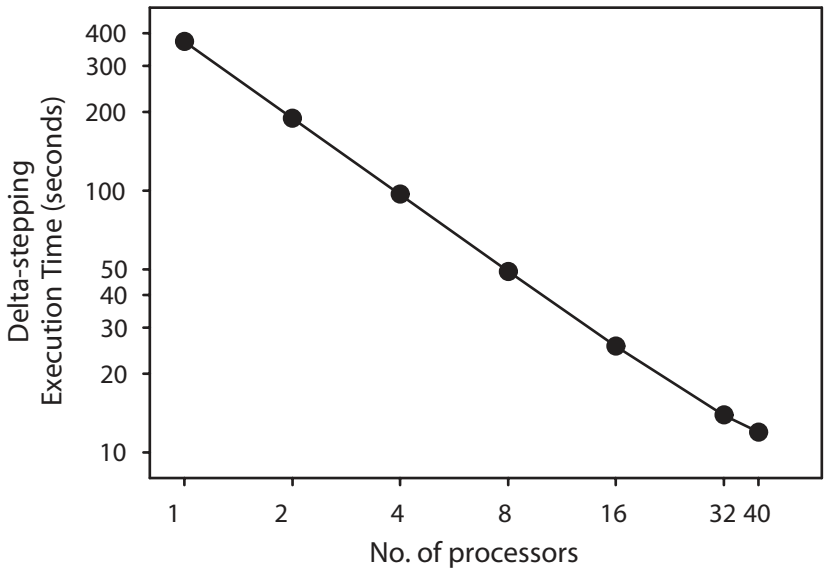


(b) USA-road-d family, Florida (FLA). 1070376 vertices, 2712798 edges.

Figure 16: A comparison of the execution time on the reference sequential platform and a single MTA-2 processor, as the bucket-width Δ is varied.

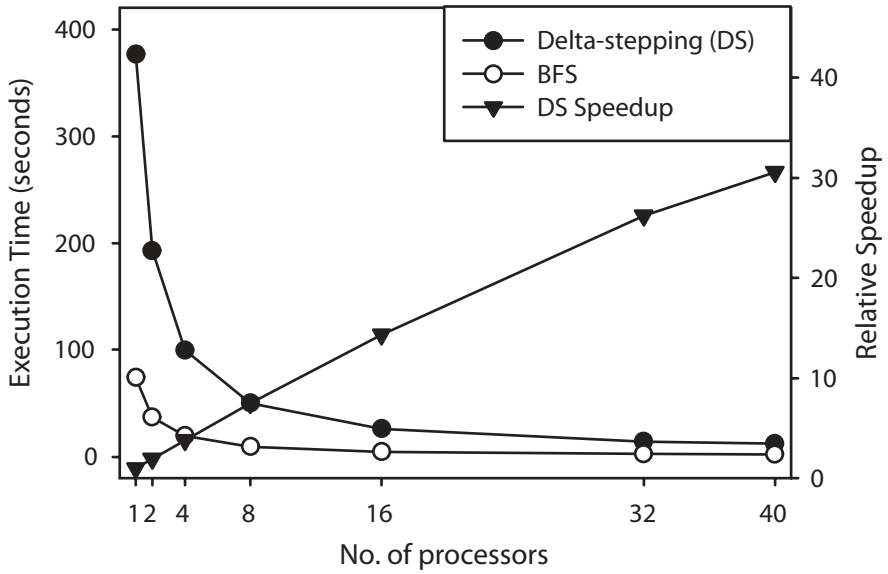


(a) Execution time and Relative Speedup (linear scale).

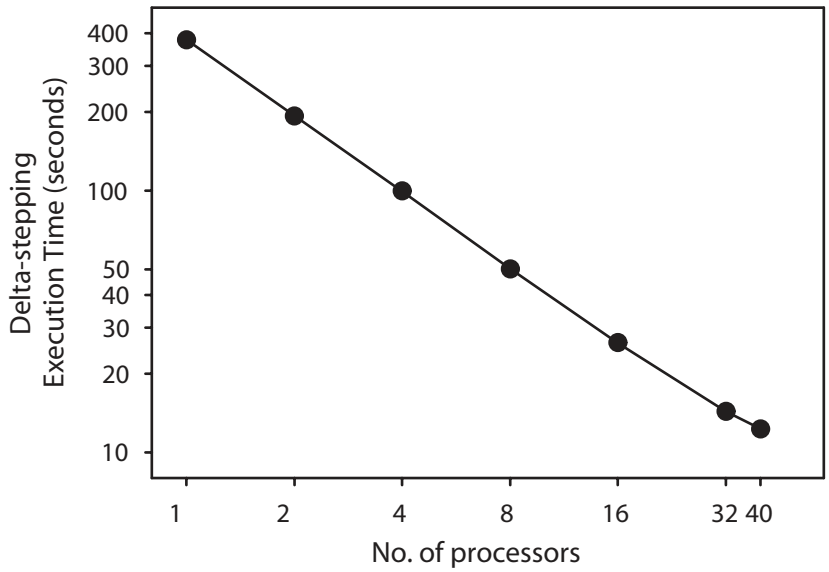


(b) Execution time vs. No. of processors (log-log scale).

Figure 17: Δ-stepping execution time and speedup on the MTA-2 for a Random4-n graph instance (directed graph, n=2^{28} vertices and $m = 4n$ edges, random edge weights).



(a) Execution time and Relative Speedup (linear scale).



(b) Execution time vs. No. of processors (log-log scale).

Figure 18: Δ-stepping execution time and speedup on the MTA-2 for a ScaleFree4-n graph instance (directed graph, n=2^{28} vertices and $m = 4n$ edges, random edge weights).

Table 15 gives the execution time of Δ-stepping on the Random4-n family, as the number of vertices is increased from 2^{21} to 2^{28}, and the number of processors is varied from 1 to 40. Observe that the relative speedup increases as the problem size is increased (for e.g., on 40 processors, the speedup for $n = 2^{21}$ is just 3.96, whereas it is 31.04 for 2^{28} vertices). This is because there is insufficient parallelism in a problem instance of size 2^{21} to saturate 40 processors of the MTA-2. As the problem size increases, the ratio of Δ-stepping execution time to multithreaded BFS running time decreases. On an average, Δ-stepping is 5 times slower than BFS for this graph family.

Table 16 gives the execution time for random graphs with a log-uniform weight distribution. With Δ set to $\frac{n}{m}$, we do a lot of additional work. The Δ-stepping to BFS ratio is typically 40 in this case, about 8 times higher than the corresponding ratio for random graphs with random edge weights. However, the execution time scales well with the number of processors for large problem sizes.

Table 17 summarizes the execution time for the Random4-C family. The maximum edge weight is varied from 4^0 to 4^{15} while keeping m and n constant. We do not notice any trend in the execution time in this case, as we normalize the edge weights to fall in the interval [0, 1]. Similarly, there is no noticeable trend in case of the Long-C family (Table 19).

Long and Square Mesh Instances

Tables 18 and 20 give the execution times for Δ-stepping on the long and square grid graphs respectively, as the problem size and number of processors are varied. For Long-n graphs with Δ set to $\frac{n}{m}$, there is insufficient parallelism to fully utilize even a single processor of the MTA-2. The execution time of the level-synchronous BFS also does not scale with the number of processors. In fact, as we see in Figure 19(a), the running time goes up in case of multiprocessor runs, as the parallelization overhead becomes significant. Also, note that the execution time on a single processor of the MTA-2 is two orders of magnitude slower than the reference sequential processor (Figure 15(b)). In case of square grid graphs (Figure 19(b)), there is sufficient parallelism to utilize up to 4 processors for a graph instance of 2^{24} vertices. For all other instances, the running time does not scale for multiprocessor runs. The ratio of the running time to BFS is about 5 in this case, and the

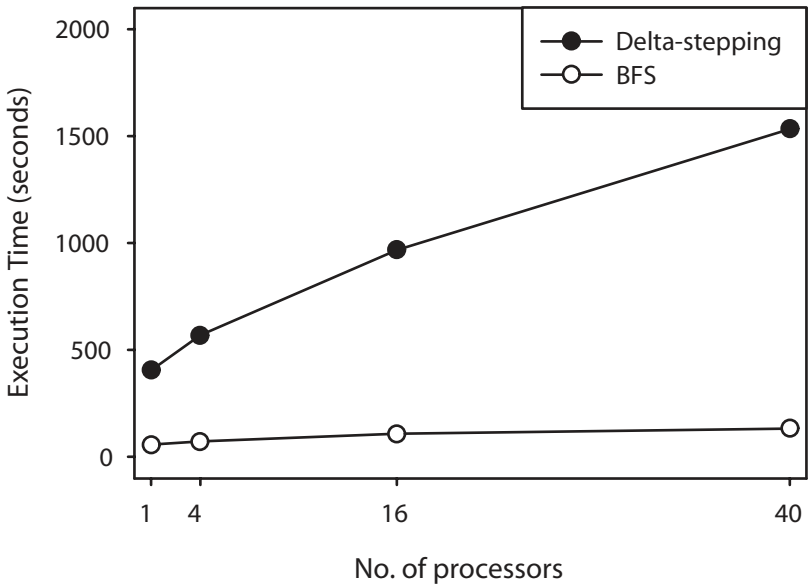
Δ-stepping MTA-2 single processor time is comparable to the sequential reference platform running time for smaller instances.

Road networks

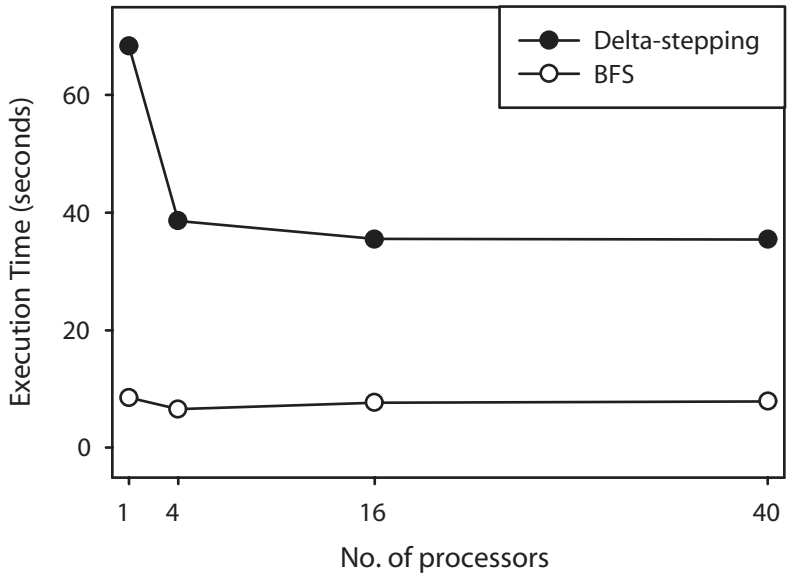
Table 21, Table 22 and Figure 20 summarize the running times on the USA and PTV Europe [156] road networks. The execution time and parallel performance is highly dependent on the value of Δ, as the normalized edge weights do not have a uniform random distribution. The behavior is best exemplified by the Europe network instance with transit time as the length function. For this graph, the maximum edge weight is 44.8458 million and the mean is 16.78 million, whereas the median weight is only 166. For $\Delta = 0.4 \approx \frac{n}{m}$, the algorithm performance tends to the worst case behavior. Now, consider the performance for Δ values of 10^{-4} and 10^{-3}. The total number of relax requests for $\Delta = 10^{-4}$ is nearly 31 million (73% more than the optimal 18 million requests), whereas for $\Delta = 10^{-3}$, the number of relax requests is 137 million (627% more than optimal). Although we do significantly more work for $\Delta = 10^{-3}$, the running time of a single MTA-2 processor is about 60 seconds, nearly 14 seconds faster than the $\Delta = 10^{-4}$ case. This is due to the MTA-2 parallelization overhead proportional to the number of parallel phases: for $\Delta = 10^{-3}$, the number of parallel phases is 10000, and for $\Delta = 10^{-4}$ it is close to 20000. We set the Δ value to the median normalized edge weight in the experiments on the full road networks. There is no significant parallel speedup, as the average relax request size per phase is low and there is insufficient parallelism in each phase to saturate multiple processors of the MTA-2.

2.5 Summary

In this chapter, we demonstrate that the massive multithreading paradigm of the Cray MTA-2 aids in the design of simple, scalable and high-performance graph algorithms. We test our BFS and *st*-connectivity implementations on large-scale real and synthetic graph instances, and report impressive results, both for algorithm execution time and parallel performance. For instance, BFS on a scale-free graph of 200 million vertices and 1 billion edges takes less than 5 seconds on a 40-processor MTA-2 system, with an absolute speedup of close to 30. We also achieve parallel speedup on the multicore Sun Niagara and the



(a) Execution time for a Long-n graph instance (directed graph, n=2^{21} vertices and $m \approx 4n$ edges, random edge weights).



(b) Execution time for a Square-n graph instance (directed graph, n=2^{24} vertices and $m \approx 4n$ edges, random edge weights).

Figure 19: Δ-stepping and BFS execution times on the MTA-2 for two mesh graph instances.

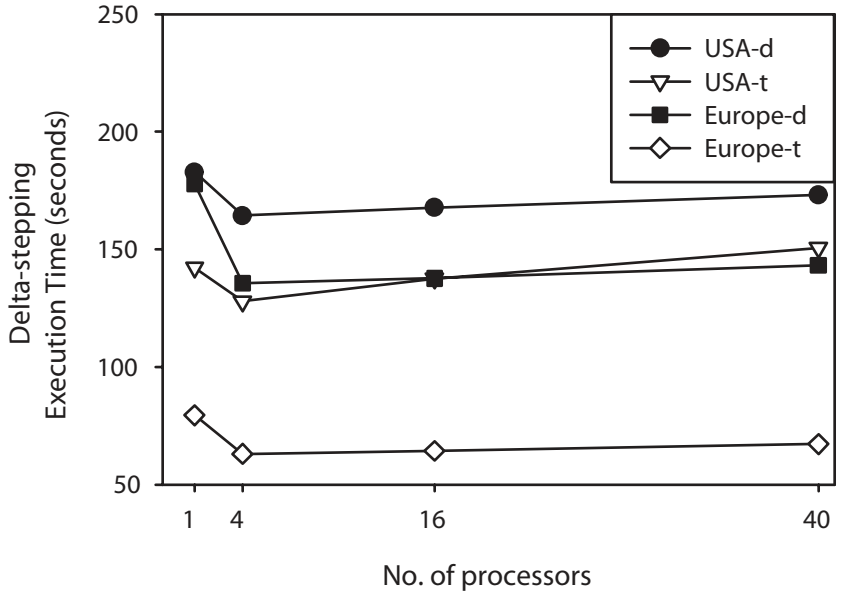


Figure 20: MTA-2 parallel performance results for the Δ-stepping algorithm on the full US and Europe road networks.

IBM p5 570 SMP, for a variety of graph instances. These are significant results in parallel computing, as prior implementations of graph algorithms report very limited or no speedup on irregular and sparse graphs, when compared to their best sequential implementations. The absolute execution time values are significant; linear-work problems involving large graphs with billions of vertices and edges can be solved in seconds on current HPC systems.

We experimentally evaluate the parallel Δ-stepping NSSP algorithm, and this work was submitted to the 9th DIMACS Shortest Paths Implementation Challenge. We study the algorithm performance for core challenge graph instances on the Cray MTA-2, and observe that our implementation execution time scales impressively with number of processors for low-diameter sparse graphs. We also analyze the performance using platform-independent Δ-stepping algorithm operation counts such as the number of *phases*, and the *request set sizes*, to explain performance across graph families. For grids and road networks, we observe that the average request set size is much smaller than corresponding low-diameter graph instances of the same size. Also, the parallelization overhead is significant for these instances, as there are a higher number of parallel phases.

We also show the dependence of the bucket-width Δ on the parallel performance of the

algorithm. For high diameter graphs, there is a trade-off between the number of phases and the amount of work done (proportional to the number of bucket insertions). The execution time is dependent on the value of Δ as well as the number of processors. In case of road networks, where the weight distribution is not uniformly random, we have to carefully choose a value of Δ to avoid doing excessive work.

Our parallel shortest path performance studies have been restricted to the Cray MTA-2 in this chapter. In future, we will extend this study to include optimized implementations of Δ-stepping on symmetric multiprocessors and multicore processors. Demonstrating scalable and efficient parallel performance for NSSP on arbitrary high-diameter graphs and road networks still remains an open challenge.

CHAPTER III

CENTRALITY ANALYSIS

Centrality analysis deals with the identification of *critical* vertices and edges in real-world graph abstractions. Graph-theoretic centrality heuristics such as betweenness and closeness are widely used in application domains ranging from social network analysis to systems biology. In this chapter, we present the *first parallel algorithms* and efficient implementations for evaluating these compute-intensive metrics. The parallel algorithms are optimized for real-world networks, and exploit topological properties such as the low-diameter and unbalanced degree distributions. These centrality implementations are integrated into our open-source SNAP (Small-world Network Analysis and Partitioning) graph framework for exploratory study and partitioning of large-scale networks. Using SNAP, we evaluate centrality indices for several large-scale networks such as web crawls, protein-interaction networks, movie-actor and patent citation networks, that are *three orders of magnitude larger* than instances that can be processed by current social network analysis packages.

The key contributions of our work specifically related to centrality analysis are as follows:

- We present the *first parallel algorithms* for efficiently computing the following centrality metrics: degree, closeness, stress, and betweenness. We optimize the algorithms to exploit typical topological features of real-world graphs.

- In Section 3.4, we present a novel approximation algorithm for *estimating* the betweenness centrality of a given vertex, for both weighted and unweighted graphs. Our approximation algorithm is based on an adaptive sampling technique that significantly reduces the number of single-source shortest path computations for vertices with high centrality. We conduct an extensive experimental study on real-world graph instances, and observe that our random sampling algorithm gives very good betweenness approximations for biological networks, road networks and web crawls.

- We present the case study of betweenness centrality analysis applied to eukaryotic protein-interaction networks (PIN). Jeong et al. [108] empirically show that betweenness is positively correlated with a protein's essentiality and evolutionary age. We observe that proteins with *high betweenness centrality but low connectivity* are abundant in the human and yeast PINs, and that current small-world network models fail to explain this finding. We discuss this case study in more detail in Section 3.6.

- As a global shortest paths-based software analysis metric, betweenness is highly correlated with routing and data congestion in information networks [100, 170]. We investigate the centrality of the integer torus, a network popularly used as the interconnection of supercomputers. We state and prove an empirical conjecture for betweenness centrality on an integer torus [10]. This result is used as a validation technique in the HPCS Graph Analysis benchmark [18], and is discussed in Section 3.7.

3.1 Centrality Metrics

Complex network analysis traces its roots to the social sciences [161, 182, 165, 75], and seminal contributions in this field date back to more than sixty years. There are several analytical tools [120, 25] for visualizing social networks, determining empirical quantitative indices, and clustering. In most applications, graph abstractions and algorithms are frequently used to help capture the salient features. Thus, social network analysis (SNA) from a graph theoretic perspective is about extracting interesting information, given a large graph constructed from a real-world dataset. Network modeling has received considerable attention in recent times, but algorithms are relatively less studied. Real-world graphs are often very large, with the number of vertices and edges ranging from several hundreds of thousands to billions. There are several key problems in large-scale network analysis that can be addressed with novel parallel algorithms and high performance implementations.

One of the fundamental problems in network analysis is to determine the *importance* or *criticality* of a particular vertex or an edge in a network. Quantifying *centrality* and *connectivity* helps us identify portions of the network that may play interesting roles. Researchers have been proposing metrics for centrality for the past 50 years, and there is no

single accepted definition. The metric of choice is dependent on the application and the network topology. Almost all metrics are empirical, and can be applied to element-level [36], group-level [63], or network-level [178] analyses. We discuss several commonly-used vertex centrality indices in this section. Note that stress and betweenness centrality have both vertex and edge formulations. We only present vertex centrality algorithms in this section – minor changes to these algorithms are necessary to compute edge centralities. SNAP includes implementations of both vertex and edge centralities.

Consider a graph $G = (V, E)$, where V is the set of vertices representing *actors* or *nodes* in the complex network, and E, the set of edges representing the relationships between the vertices. The number of vertices and edges are denoted by n and m respectively. The graphs can be directed or undirected. We will assume that each edge $e \in E$ has a positive integer weight $w(e)$. For unweighted graphs, we use $w(e) = 1$. A *path* from vertex s to t is defined as a sequence of edges $\langle u_i, u_{i+1} \rangle$, $0 \leq i \leq l$, where $u_0 = s$ and $u_l = t$. The *length* of a path is the sum of the weights of edges. We use $d(s, t)$ to denote the distance between vertices s and t (the minimum length of any path connecting s and t in G). Let us denote the total number of shortest paths between vertices s and t by σ_{st}, and the number passing through vertex v by $\sigma_{st}(v)$.

Degree Centrality

The degree centrality DC of a vertex v is simply the degree $deg(v)$ for undirected graphs. For directed graphs, we can define two variants: in-degree centrality and out-degree centrality. This is a simple local measure, based on the notion of neighborhood. This index is useful in case of static graphs, for situations when we are interested in finding vertices that have the most direct connections to other vertices.

Closeness Centrality

This index measures the closeness, in terms of *distance*, of a vertex to all other vertices in the network. Vertices with a smaller total distance are considered more important. Several closeness-based metrics [27, 163, 149] have been developed by the SNA community. A

commonly used definition is the reciprocal of the total distance from a particular vertex to all other vertices:

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

Unlike degree centrality, this is a global metric. To calculate the closeness centrality of a vertex v, we may apply breadth-first search (BFS, for unweighted graphs) or a single-source shortest path (SSSP, for weighted graphs) algorithms from v. Note that the closeness centrality of a single vertex can be determined in linear time.

Stress Centrality

Stress centrality is a metric based on shortest paths counts, first presented in [169]. It is defined as

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

Intuitively, this metric deals with the *work* done by each vertex in a communications network. The number of shortest paths that contain an element v will give an estimate of the amount of stress a vertex v is under, assuming communication will be carried out through shortest paths all the time. This index can be calculated using a variant of the all-pairs shortest-paths algorithm, that calculates and stores all shortest paths between any pair of vertices.

Betweenness Centrality

Betweenness centrality is another shortest paths enumeration-based metric, introduced by Freeman in [74]. Let $\delta_{st}(v)$ denote the *pairwise dependency*, or the fraction of shortest paths between s and t that pass through v:

$$\delta_{st}(v) \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Betweenness centrality of a vertex v is defined as

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

This metric can be thought of as *normalized* stress centrality. Betweenness centrality of a vertex measures the control a vertex has over communication in the network, and can be used to identify critical vertices in the network. High centrality indices indicate that a vertex can reach other vertices on relatively short paths, or that a vertex lies on a considerable fraction of shortest paths connecting pairs of other vertices.

This index has been extensively used in recent years for analysis of social as well as other large scale complex networks. Some applications include biological networks [108, 155, 54], study of sexual networks and AIDS [129], identifying key actors in terrorist networks [121, 46], organizational behavior [39], supply chain management [44], and transportation networks [89].

There are a number of commercial and research software packages for SNA (e.g., Pajek [25], InFlow [120], UCINET [8]) which can also be used to determine these centrality metrics. However, they can only process comparatively small networks (in most cases, sparse graphs with less than 40,000 vertices). Our goal is to develop fast, high-performance implementations of these metrics so that one can analyze large-scale real-world graphs of millions to billions of vertices.

3.1.1 Betweenness Centrality: Sequential Algorithm

In this section, we discuss sequential algorithms to compute exact betweenness centrality. Unlike closeness centrality, there is no known algorithm to compute betweenness centrality of a single vertex in linear time.

A straightforward way of computing betweenness centrality for each vertex would be as follows:

1. compute the length and number of shortest paths between all pairs (s, t).
2. for each vertex v, calculate every possible pair-dependency $\delta_{st}(v)$ and sum them up.

The complexity is dominated by step 2, which requires $\Theta(n^3)$ time summation and $\Theta(n^2)$ storage of pair-dependencies. Popular SNA tools like UCINET use an adjacency matrix to store and update the pair-dependencies. This yields an $\Theta(n^3)$ algorithm for betweenness

by augmenting the Floyd-Warshall algorithm for the all-pairs shortest-paths problem with path counting [31].

Alternately, we can modify Dijkstra's single-source shortest paths algorithm to compute the pair-wise dependencies. Observe that a vertex $v \in V$ is on the shortest path between two vertices $s, t \in V$, *iff* $d(s,t) = d(s,v) + d(v,t)$. Define a set of *predecessors* of a vertex v on shortest paths from s as $pred(s,v)$. Now each time an edge $\langle u, v \rangle$ is scanned for which $d(s,v) = d(s,u) + d(u,v)$, that vertex is added to the predecessor set $pred(s,v)$. Then, the following relation would hold:

$$\sigma_{sv} = \sum_{u \in pred(s,v)} \sigma_{su}$$

Setting the initial condition of $pred(s,v) = s$ for all neighbors v of s, we can proceed to compute the number of shortest paths between s and all other vertices. The computation of $pred(s,v)$ can be easily integrated into Dijkstra's SSSP algorithm for weighted graphs, or Breadth-First Search for unweighted graphs. But even in this case, determining the fraction of shortest paths using v, or the pair-wise dependencies $\delta_{st}(v)$, proves to be the dominant cost. The number of shortest $s - t$ paths using v is given by $\sigma_{st}(v) = \sigma_{sv} \cdot \sigma_{vt}$. Thus computing $BC(v)$ requires $O(n^2)$ time per vertex v, and $O(n^3)$ time in all. This algorithm is the most commonly used one for evaluating betweenness centrality.

To exploit the sparse nature of typical real-world graphs, Brandes [31] designed an algorithm that computes the betweenness centrality score for all vertices in the graph in $O(mn + n^2 \log n)$ time for weighted graphs, and $O(mn)$ time for unweighted graphs. The main idea is as follows. We define the *dependency* of a source vertex $s \in V$ on a vertex $v \in V$ as

$$\delta_s(v) = \sum_{t \in V} \delta_{st}(v)$$

The betweenness centrality of a vertex v can be then expressed as $BC(v) = \sum_{s \neq v \in V} \delta_s(v)$. The dependency $\delta_s(v)$ satisfies the following recursive relation:

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

The algorithm is now stated as follows. First, n SSSP computations are done, one for each $s \in V$. The predecessor sets $pred(s,v)$ are maintained during these computations. Next, for

every $s \in V$, using the information from the shortest paths tree and predecessor sets along the paths, compute the dependencies $\delta_s(v)$ for all other $v \in V$. To compute the centrality value of a vertex v, we finally compute the sum of all dependency values. The $O(n^2)$ space requirements can be reduced to $O(m+n)$ by maintaining a *running centrality score*.

3.2 Exact Parallel Centrality Algorithms

In this section, we present novel parallel algorithms to compute the various centrality metrics, optimized for real-world networks. To our knowledge, these are the first parallel algorithms for centrality analysis. We exploit the typical low-diameter (small-world) property to reveal an additional level of parallelism in graph traversal, and take the unbalanced degree distribution into consideration while designing algorithms for the shortest-path based enumeration metrics. We demonstrate that these algorithms perform well in practice, and our implementations are freely available as part of the SNAP framework.

3.2.1 Degree Centrality

We store the in- and out-degree of each vertex during construction of the graph abstraction. Thus, determining the degree centrality of a particular vertex is a constant-time lookup operation. As noted previously, degree centrality (or vertex-connectivity) is a useful local metric and probably the most studied measure in complex network analysis.

3.2.2 Closeness Centrality

Closeness centrality of a vertex v can be computed by a simple breadth-first traversal from v (single-source shortest paths in case of weighted graphs), and requires no auxiliary data structures. Thus vertex centrality computation can be done in parallel by a straightforward parallelization of BFS. In a typical network analysis scenario, we would require centrality scores of all the vertices in the graph. We can then compute n BFS (shortest path) trees in parallel, one for each vertex $v \in V$. On p processors, this would yield $T_C = O(\frac{nm+n^2}{p})$ and $T_M = \frac{nm}{p}$ for unweighted graphs. For weighted graphs, using a naïve queue-based representation for the expanded frontier, we can compute all the centrality metrics in $T_C = O(\frac{nm+n^3}{p})$ and $T_M = \frac{2nm}{p}$. The bounds can be further improved with the

use of efficient priority queue representations.

Evaluating closeness centrality of all the vertices in a graph is computationally intensive; hence, it is valuable to investigate approximate algorithms. Using a random sampling technique, Eppstein and Wang [67] show that the closeness centrality of all vertices in a weighted, undirected graph can be approximated with high probability in $O(\frac{\log n}{\epsilon^2}(n \log n + m))$ time, and an additive error of at most $\epsilon\Delta_G$ (ϵ is a fixed constant, and Δ_G is the diameter of the graph). The algorithm proceeds as follows. Let k be the number of iterations needed to obtain the desired error bound. In iteration i, pick vertex v_i uniformly at random from V and solve the SSSP problem with v_i as the source. The estimated centrality is given by

$$\tilde{CC}(v) = \frac{k}{n\Sigma_{i=1}^{k} d(v_i, u)}$$

The error bounds follow from a result by Hoeffding [99] on probability bounds for sums of independent random variables.

We design a parallel algorithm for approximate closeness centrality as follows. Each processor runs SSSP computations from $\frac{k}{p}$ vertices and stores the evaluated distance values. The cost of this step is given by $T_C = O(\frac{k(m+n)}{p})$ and $T_M = \frac{km}{p}$ for unweighted graphs. For real-world graphs, the number of sample vertices k can be set to $\Theta(\frac{\log n}{\epsilon^2})$ to obtain the error bounds given above. The approximate closeness centrality value of each vertex can then be calculated in $O(k) = O(\frac{\log n}{\epsilon^2})$ time, and the summation for all n vertices would require $T_C = O(\frac{n \log n}{p\epsilon^2})$ and constant T_M.

3.2.3 Stress and Betweenness Centrality

Computing stress and betweenness centrality is more involved than closeness centrality. These two metrics require all-pairs shortest path enumeration, and there is no known algorithm to compute the betweenness/stress index of a single vertex or edge in linear time. We design two novel parallel algorithms for vertex betweenness centrality that are particularly suited for real-world sparse graphs. Alg. 7 outlines the general approach in the case of unweighted graphs. On each BFS computation from s, the queue Q stores the current set of vertices to be visited, S contains all the vertices reachable from s, and $P(v)$ is the predecessor set associated with each vertex $v \in V$. The arrays d and σ store the distance

Algorithm 7: Parallel betweenness centrality for unweighted graphs.

Input: $G(V, E)$

Output: Array $BC[1..n]$, where $BC[v]$ gives the centrality metric for vertex v

```
1  for all v ∈ V in parallel do
2      BC[v] ← 0;
   for all s ∈ V in parallel do
3      S ← empty stack;
4      P[w] ← empty list, w ∈ V;
5      σ[t] ← 0, t ∈ V; σ[s] ← 1;
6      d[t] ← −1, t ∈ V; d[s] ← 0;
7      Q → empty queue;
8      enqueue s ← Q;
9      while Q not empty do
10         dequeue v ← Q;
11         push v → S;
12         for each neighbor w of v in parallel do
13             if d[w] < 0 then
14                 enqueue w → Q;
15                 d[w] ← d[v] + 1;
16             if d[w] = d[v] + 1 then
17                 σ[w] ← σ[w] + σ[v];
18                 append v → P[w];
19     δ[v] ← 0, v ∈ V;
20     while S not empty do
21         pop w ← S;
22         for v ∈ P[w] do
23             δ[v] ← δ[v] + (σ[v]/σ[w])(1 + δ[w]);
24         if w ≠ s then
25             BC[w] ← BC[w] + δ[w];
```

from s, and shortest path counts, respectively. The centrality values are computed in steps 22–25, by summing the dependencies $\delta(v)$, $v \in V$. The final scores need to be divided by two if the graph is undirected, as all the shortest paths are counted twice.

We observe that parallelism can be exploited at two levels:

- The BFS/SSSP computations from each vertex can be done concurrently, provided the centrality running sums are updated atomically.
- The actual BFS/SSSP can be also be parallelized. When visiting the adjacencies of a vertex, edge relaxation can be done concurrently.

We will refer to the parallelization approach that concurrently computes the shortest path trees (steps 3–25 in Alg. 7) as the *coarse-grained* parallel betweenness centrality algorithm, and the latter approach in which a single BFS/SSSP traversal is parallelized, as the *fine-grained* algorithm.

There are performance trade-offs associated with both these algorithms when implemented on parallel systems. The coarse-grained algorithm assigns each processor a fraction of the vertices from which to initiate SSSP computations. The vertices can be assigned dynamically to processors, so that work is distributed as evenly as possible. For this approach, graph traversal requires no synchronization, and the centrality metrics can be computed exactly provided they are accumulated atomically (step 25 in Alg. 7). Alternately, each processor can store its partial sum of the centrality score for every vertex, and all the sums can be merged using an efficient global reduction operation. However, the problem with the coarse-grained algorithm is that the auxiliary data structures – the stack S, list of predecessors P, and the BFS queue Q – need to be replicated on each processor for doing concurrent traversals. The memory requirements scale as $O(p(m+n))$, and this approach becomes infeasible for large-scale graphs.

In the fine-grained algorithm, we parallelize each BFS/SSSP computation, and the memory requirement is $O(m+n)$. Note that every centrality computation iteration is composed of two phases, the BFS tree computation (steps 3–18), and accumulation of centrality scores (steps 19–24). In previous work, we discuss algorithms and efficient implementations for

fine-grained parallel BFS [14] and single-source shortest paths [134, 51]. These algorithms can be directly applied for parallelizing the traversal phase. In the subsequent phase, we need to visit vertices in the non-increasing order of their distance from the source vertex. Real-world social and technological networks typically demonstrate the small-world property, i.e., the graph diameter is usually a constant value, or in some cases $O(\log n)$. Thus, there will be a significant number of vertices at a given depth from the source vertex, and the centrality scores of all these vertices can be accumulated in parallel. We discuss implementation details and performance of both the algorithms in Section 3.3.3.

3.2.4 Optimizations for real-world graphs

In the design of fine-grained algorithms for betweenness and stress centrality, we achieve concurrency in graph traversal by exploiting the small-world nature (no long paths, low graph diameter) of real-world networks. The unbalanced degree distribution is another important graph characteristic we need to consider while optimizing centrality algorithms. It has been observed that real networks tend to have highly skewed degree distributions that can be approximated by power-laws in some cases. We see a significant number of low-degree vertices in such networks, and a comparatively smaller number of vertices of very high degree (can be as large as $O(n)$). Centrality is also correlated with degree: intuitively, high-degree vertices may have high centrality scores as a significant number of shortest paths pass through them. The degree distribution has very little effect on the performance of coarse-grained parallel centrality algorithms, as we do a full graph traversal on every outer-loop iteration. Each iteration roughly takes the same time, and even a static distribution of work among processors is reasonably balanced. However, while designing fine-grained centrality algorithms, we need to explicitly consider unbalanced degree distributions. In a level-synchronized parallel BFS where vertices are statically assigned to processors without considering their degree, it is highly probable that there will be phases with severe work imbalance. For instance, consider the case in which one processor is assigned a group of low-degree vertices, and another processor has to expand the BFS frontier from a set of high-degree vertices (say, degree $O(n)$). In the worst case, we will not achieve any parallel

speedup using this approach. Our fine-grained BFS and shortest path algorithms [14, 134] are designed to be independent of degree distribution, and we use these optimized algorithms as the inner routines for the fine-grained centrality algorithms.

We can cut down on the space requirements of the algorithms with appropriate data structure choices. Observe that in Alg. 7, Q is not needed, as the BFS frontier vertices are also added to S. Also, note that the size of the list of predecessors of a vertex is bounded by the degree (in-degree for directed graphs), and $|\Sigma_{v \in V} P_s(v)| = O(m)$, which is an upper bound on the predecessor list space requirements. Thus, we have an estimate of predecessor list sizes for all the vertices in the graph, and memory allocation can be done as a preprocessing step to the actual centrality metric computation. We can use different data structures for representing the predecessor lists: for low-degree vertices (say, degree less than 10), we can use bit vectors to identify adjacencies that are in the predecessor set; for high-degree vertices, we use cache-friendly dynamic adjacency arrays.

There are several other optimizations that are applicable to special graphs. If a graph is composed of several large connected components, we can run the linear-time connected components algorithm to preprocess the network and identify the components. The centrality indices of the various components can then be evaluated concurrently.

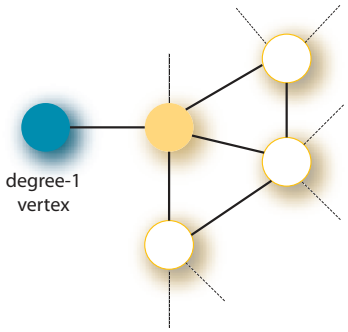


Figure 21: The betweenness centrality index of a degree-1 vertex is 0. We need not traverse the graph from a degree-1 vertex if we store the shortest path tree from its adjacency.

Observe that by definition, the betweenness centrality score of a degree-1 vertex is zero. Also, we do not need to traverse the graph from a degree-1 vertex (steps 3–18 of Alg. 7), if we already have the shortest-path tree from its adjacent vertex v_a. Alg. 8 gives the revised algorithm for the centrality accumulation stage from a degree-1 vertex. Note

Algorithm 8: Betweenness centrality accumulation and dependency computation stage for degree-1 vertices.

Input: degree-1 vertex v_s, adjacent vertex v_a, stack of vertices visited from v_a: S, auxiliary information from graph traversal from v_a: σ and P

Output: Updated betweenness centrality scores

```
1 δ[v] ← 0, v ∈ V;
2 P[v_s] ← empty list;
3 while S not empty do
4 |   pop w ← S;
5 |   for v ∈ P[w] do
6 |   |   δ[v] ← δ[v] + (σ[v]/σ[w])(1 + δ[w]);
7 |   BC[w] ← BC[w] + δ[w];
```

that the centrality value increments for all the vertices except v_a are identical to the score increments corresponding to a traversal from v_a. In addition, the centrality score of v_a is incremented in this step, and the predecessor list of v_s is cleared. Thus, we can further simplify the algorithm by moving all the computation done in the v_s iteration to the v_a iteration. Alg. 9 gives the revised procedure for the centrality accumulation stage from the vertex adjacent to a degree-1 vertex. This supercedes Alg. 8 and we do not require any computation from degree-1 vertices (steps 3–25 in the original betweenness centrality Alg. 7). This optimization is particularly effective in networks such as the web-graph and protein-interaction networks where there are a significant percentage of degree-1 vertices (unannotated proteins with few interactions; web pages linking to a hub-site).

The degree-1 betweenness centrality optimization can be generalized as follows. We *reuse the shortest path tree* from the vertex adjacent to a degree-1 vertex to cut down on computation. We can extend this idea to an arbitrary vertex in the graph, if we know the shortest path trees from all its adjacencies. So every vertex in the graph has to either belong to the set of vertices with precomputed shortest path information, or have all its adjacencies in this set. For undirected graphs, the minimal set of vertices for which we need to store the shortest path trees from becomes its *vertex cover*, a known NP-complete problem [79, 49].

Lemma 1 (Bellman Criterion)**.** *A vertex $v \in V$ lies on a shortest path between vertices*

Algorithm 9: Augmenting the betweenness centrality accumulation stage from the adjacency of a degree-1 vertex (supercedes Alg. 8).

Input: Vertex v_a with d_1 degree-1 adjacencies. S, P, σ computed in the graph traversal stage from v_a.

Output: Updated betweenness centrality scores

```
1  δ[v] ← 0, v ∈ V;
2  P[v_s] ← 0, (v, v_s) ∈ E and degree(v_s) = 1;
3  while S not empty do
4      pop w ← S;
5      for v ∈ P[w] do
6          δ[v] ← δ[v] + (σ[v]/σ[w])(1 + δ[w]);
7      if w ≠ v_a then
8          BC[w] ← BC[w] + (d_1 + 1)δ[w];
       else
9          BC[w] ← BC[w] + d_1 δ[w];
```

$s, t \in V$, *if and only if* $d(s,t) = d(s,v) + d(v,t)$.

Theorem 1. *For an unweighted, undirected graph $G(V,E)$, given the shortest path tree information from vertices in the vertex cover, we can construct the shortest path tree from any vertex in the graph, without re-traversing the graph.*

Proof. Consider a vertex $s \notin V_c$, where V_c is the vertex cover of the graph. Then all the vertices adjacent to s belong to the vertex cover. For an arbitrary vertex $v \in V$, if $v \in V_c$, we know that $d_s(v) = d_v(s)$. Otherwise, we have $d_s(v) = min(d_s(v) + d_u(v)), (v,u) \in E$, since any path to v has to pass though one of the adjacencies of s. Thus we have the distance values of all the vertices in the graph. We do not need to explicitly compute the shortest path-count array σ and the predecessor lists $P[i]$ for this vertices. Let $u_1, u_2, ..., u_k$ be the adjacencies of s for which $d_s(v) = min(d_s(v) + d_{u_i}v), 1 \leq i \leq k$. Then from the Bellman criterion, we have $\sigma_s(v) = \Sigma_{i=1}^{k}(\sigma_s(u_i)\sigma_{u_i}(v))$, and $P_s v = \cup_i P_{u_i}$. This information need not be stored, but computed on-the-fly in the betweenness centrality accumulation phase.

An important application of vertex and edge centrality is community identification in real-world networks. The Girvan-Newman [82] algorithm iteratively partitions a network by identifying high betweenness centrality edges, removing them and recomputing centrality scores. For unweighted, undirected graphs, the worst case complexity of the sequential

algorithm is $O(mn^2)$ and so exact computation becomes infeasible for large-scale networks. One simple technique to the reduce the graph size is to prune unimportant vertices and edges. For instance, degree-1 vertices can be pruned since their centrality scores are zero. Generalizing this observation, we can extract biconnected components of the network first as a preprocessing step and individually cluster the components. We discuss a parallel algorithm for community identification based on centrality estimation in Chapter 4.

3.2.5 Parallel Implementation

We use a cache-friendly, $O(m + n)$-space adjacency array representation [152] for storing the graph. This ensures minimal overhead for iterating over the adjacencies of a particular vertex in the graph, and increases spatial locality in graph traversal. We implement two algorithms for all the centrality metrics: a fine-grained version that parallelizes graph traversal, and a coarse-grained version that does concurrent graph traversals from various processors. In this section, we will primarily focus on performance results of betweenness centrality. The implementations include optimizations discussed in Section 3.2.4.

Our implementations are optimized for multicore processors, symmetric multiprocessors, and multithreaded architectures. The SMP and multicore codes are portable to various systems, but the multithreaded implementation is optimized for the Cray MTA-2 system. The MTA-2 is a high-end shared memory system offering two unique features that aid considerably in the design of irregular algorithms: fine-grained parallelism and low-overhead word-level synchronization. It has no data cache; rather than using a memory hierarchy to reduce latency, the MTA-2 processors use hardware multithreading to tolerate the latency. The word-level synchronization support complements multithreading and makes performance primarily a function of parallelism. Since graph algorithms have an abundance of parallelism, yet often are not amenable to partitioning, the MTA-2 architectural features lead to superior performance and scalability.

For computing centrality metrics on weighted graphs, we use a fine-grained parallel single-source shortest path [134] algorithm as the inner routine. However, the centrality accumulation step cannot be easily parallelized, as it requires visiting adjacencies in the

order of non-increasing distance from the source vertex. We can only exploit parallelism in this step if the maximum edge weight $C << n$. The coarse-grained algorithm is straightforward and just requires minor changes to Alg. 7.

3.3 Exact Betweenness Computation Experimental Study

3.3.1 Platforms

We report multithreaded performance results on a 40-processor Cray MTA-2 system with 160 GB uniform shared memory. Each processor has a clock speed of 220 MHz and support for 128 hardware threads. The code is written in C with MTA-2 specific pragmas and directives for parallelization. We compile the code using the MTA-2 C compiler (Cray Programming Environment (PE) 2.0.3) with -O3 and -par flags. The MTA-2 code also compiles and runs on sequential processors without any modification.

Our test platform for the SMP implementations is an IBM Power 570 server. The IBM Power 570 is a 16-way symmetric multiprocessor with 16 1.9 GHz Power5 cores with simultaneous multithreading (SMT), 32 MB shared L3 cache, and 256 GB shared memory. The code is compiled using the IBM XL C compiler v7.0 with the -O3 optimization flag.

We report multicore system performance results on the Sun Fire T2000 server, with the Sun UltraSPARC T1 (Niagara) processor. This system has eight cores running at 1.0 GHz, each of which is four-way multithreaded. There are eight integer units with a six-stage pipeline on chip, and four threads running on a core share the pipeline. The cores also share a 3 MB L2 cache, and the system has a main memory of 16 GB. There is only one floating point unit (FPU) for all cores. We compile our codes with the Sun C compiler v5.8 and the flags -xtarget=ultraT1 -xarch=v9b -xopenmp.

3.3.2 Problem Instances

We test our centrality metric implementations on a variety of real-world graphs, summarized in Table 1. Our implementations have been extended to read input files in both PAJEK and UCINET graph formats. We also use a synthetic graph generator [41] to generate graphs with small-world characteristics and unbalanced degree distributions. The degree distributions of the IMDB test graph instance is shown in Figure 22. We observe that the

Table 1: Networks used in the exact betweenness centrality computation experimental study.

Dataset	Source	Network description
ND-actor	[20]	an undirected graph of 392,400 vertices (movie actors) and 31,788,592 edges. An edge corresponds to a link between two actors, if they have acted together in a movie. The dataset includes actor listings from 127,823 movies.
ND-web	[20]	a directed network with 325,729 vertices and 1,497,135 arcs. Each vertex represents a web page within the Univ. of Notredame *nd.edu* domain, and the arcs represent from $\rightarrow$ to links.
ND-yeast	[20]	undirected network with 2114 vertices and 2277 edges. Vertices represent proteins, and the edges represent interactions between them in the yeast network.
UMD-human	[16]	undirected network with 18669 vertices and 43568 edges. Vertices represent proteins, and the edges represent interactions between them in the human interactome.
PAJ-patent	[26]	a network of about 3 million U.S. patents granted between January 1963 and December 1999, and 16 million citations made among them between 1975 and 1999.
PAJ-cite	[26]	the *Lederberg* citation dataset, produced using HistCite, in PAJEK graph format with 8843 vertices and 41609 edges.

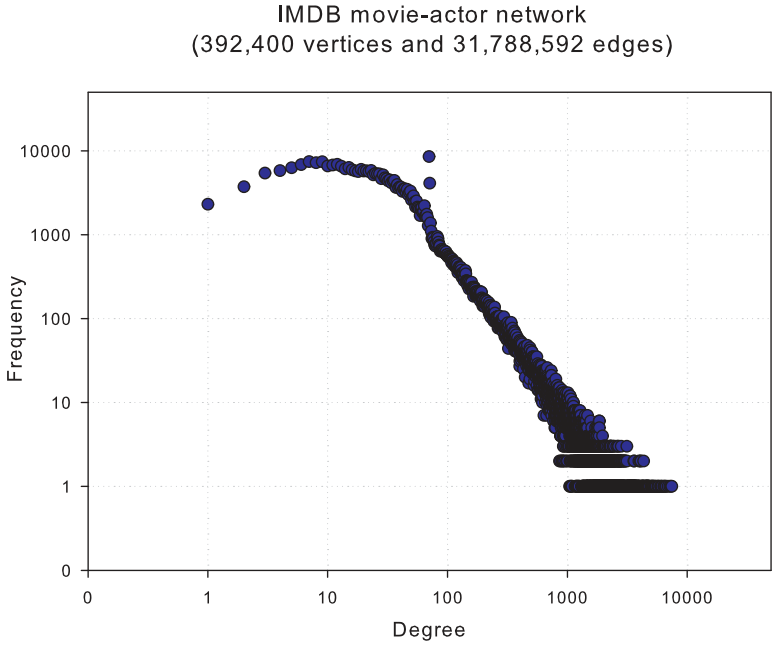


Figure 22: Vertex degree distributions of the IMDB movie-actor network used in the exact betweenness computation experimental study.

degree distributions of most of the networks are unbalanced with a heavy tail, which is in agreement with prior experimental studies.

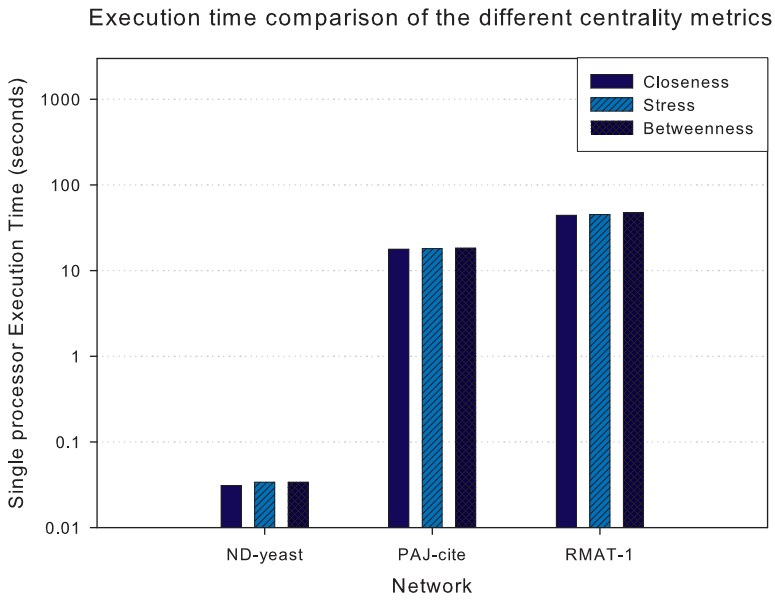
3.3.3 Results and Analysis

Figure 23 compares the single processor execution time of closeness, betweenness and stress centrality for three networks of different sizes, on the MTA-2 and the Power 570. All three metrics are of the same computational complexity and exhibit nearly similar running times in practice.

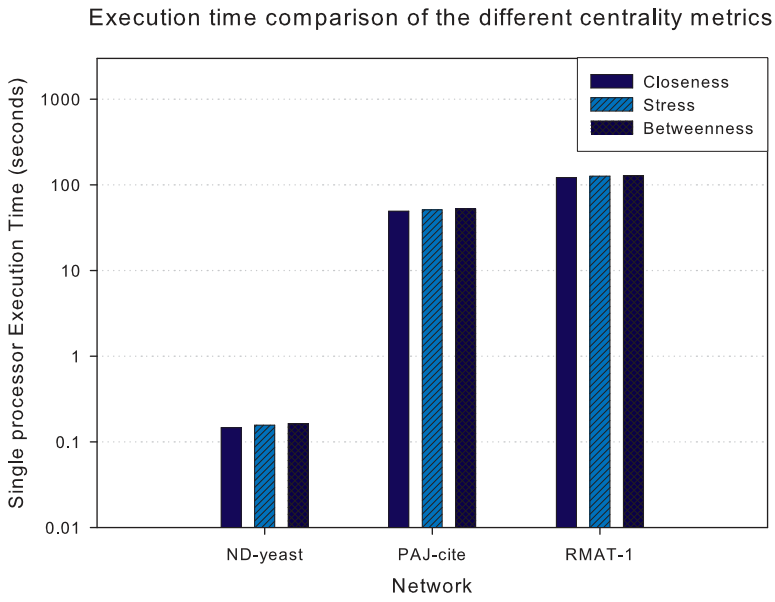
The MTA-2 performance results are for the fine-grained centrality implementations. On SMPs, the coarse-grained version outperforms the fine-grained algorithm on current systems due to the parallelization and synchronization overhead involved in the fine-grained version. We only have a modest number of processors on current SMP systems, so each processor can run a concurrent shortest path computation and create auxiliary data structures. On our target system, the Power 570, we can compute centrality metrics for graphs with up to 100 million edges using this coarse-grained implementation.

Figure 24 and Figure 25 summarize multiprocessor execution times for computing betweenness centrality on the Power 570 and the MTA-2. Figure 24(a) gives the running times for the ND-actor graph on the Power 570 and the MTA-2. As expected, the execution time scales nearly linearly with the number of processors. It is possible to evaluate the centrality metric for the entire ND-actor network in 42 minutes, on 16 processors of the Power 570. We observe similar performance for the patents citation data. This includes the optimizations for undirected, unweighted real-world networks discussed in Section 3.2.4.

Figs. 25(a) and 25(b) plot the execution time on the MTA-2 and the Power 570 for ND-web, and a synthetic graph instance of the same size generated using the R-MAT algorithm. Note that the actual execution time is dependent on the graph structure; for the same problem size, the synthetic graph instance takes much longer than the ND-web graph. The web crawl is a directed network, and the strongly connected components and degree-1 optimizations help in significantly reducing the execution time.

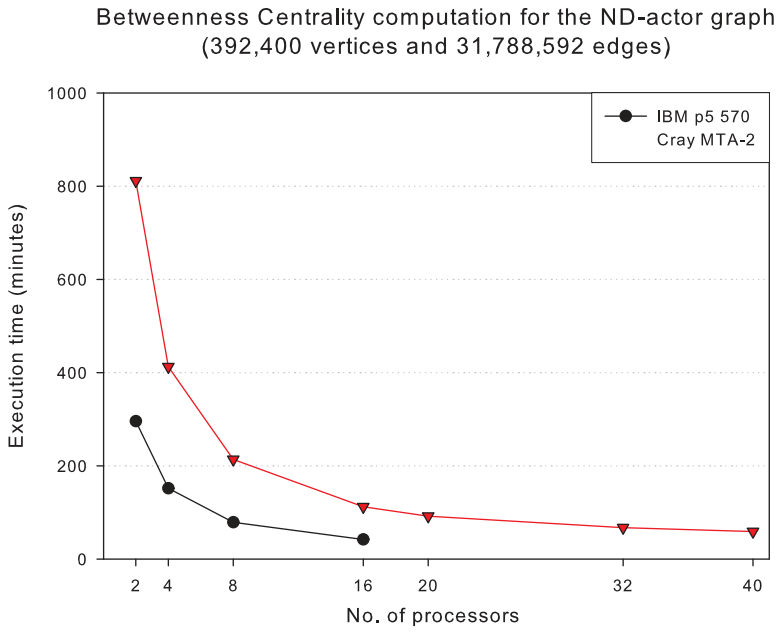


(a)

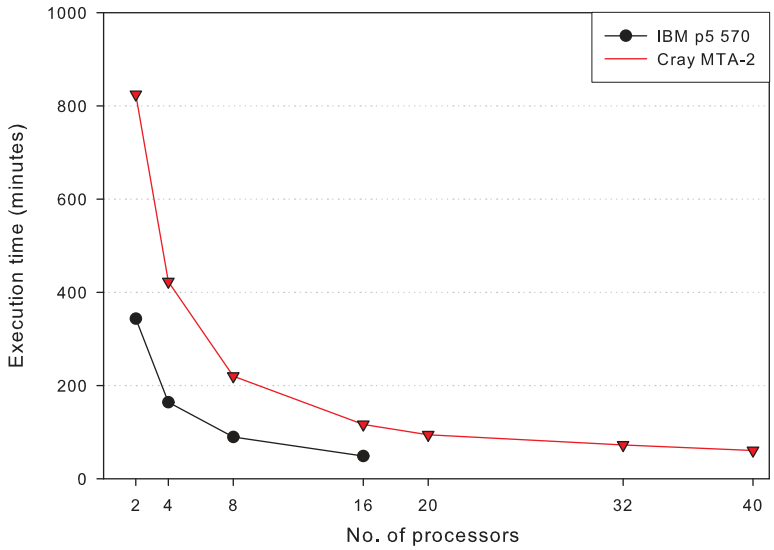


(b)

Figure 23: Single processor execution time comparison of the centrality metric implementations on the IBM Power 570 (left) and the Cray MTA-2 (right).

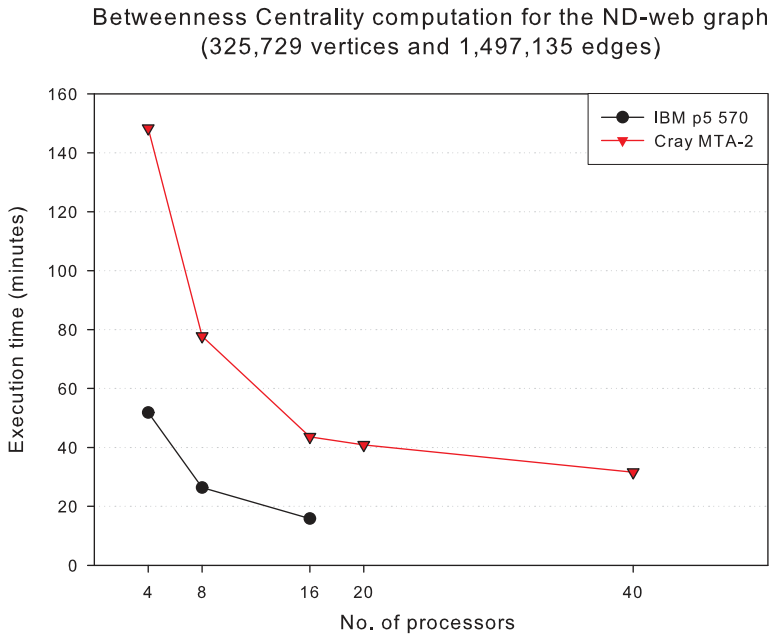


(a)

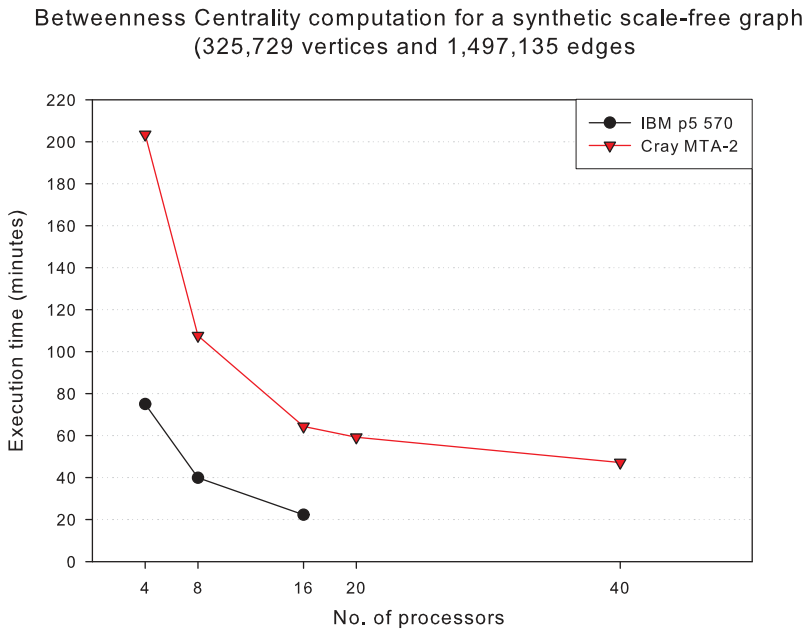


(b)

Figure 24: Parallel performance of exact betweenness centrality computation for various graph instances on the Power 570 and the MTA-2.



(a)



(b)

Figure 25: Parallel performance of exact betweenness centrality computation for various graph instances on the Power 570 and the MTA-2.

3.4 Approximating Betweenness

The Betweenness centrality measure is based on shortest path computations, and is widely used in complex network analysis. As we discussed in the previous section, it is computationally-expensive to exactly determine betweenness; currently the fastest-known sequential algorithm requires $O(nm)$ time for unweighted graphs and $O(nm + n^2 \log n)$ time for weighted graphs. These are also the worst-case time bounds for computing the betweenness score of a single vertex. We present a novel approximation algorithm for computing betweenness centrality of a given vertex, for both weighted and unweighted graphs. Our approximation algorithm is based on an adaptive sampling technique that significantly reduces the number of single-source shortest path computations for vertices with high centrality. We conduct an extensive experimental study on real-world graph instances, and observe that our random sampling algorithm gives very good betweenness approximations for biological networks, road networks and web crawls.

Fast centrality estimation is an important problem, as a good approximation would be an acceptable alternative to exact scores. Currently the fastest exact algorithms for shortest path enumeration-based metrics require n shortest-path computations; however, it is possible to estimate centrality by extrapolating scores from a fewer number of path computations. Using a random sampling technique, Eppstein and Wang [67] show that the closeness centrality of all vertices in a weighted, undirected graph can be approximated with high probability in $O(\frac{\log n}{\epsilon^2}(n \log n + m))$ time, and an additive error of at most $\epsilon\Delta_G$ (ϵ is a fixed constant, and Δ_G is the diameter of the graph). However, betweenness centrality scores are harder to estimate, and the quality of approximation is found to be dependent on the vertices from which the shortest path computations are initiated from (here, we will refer to them as the set of *source vertices* for the approximation algorithm). Recently, Brandes and Pich [34] presented centrality estimation heuristics, where they experimented with different strategies for selecting the source vertices. They observe that a random selection of source vertices is superior to deterministic strategies.

While prior approaches approximate centrality scores of *all* vertices in the graph, there are no known algorithms to compute the centrality of a single vertex in time faster than

computing the betweenness of all vertices.We present a novel *adaptive sampling*-based algorithm for approximately computing betweenness centrality of a given vertex. Our primary result is as follows:

For $0 < \epsilon < 0.5$, if the centrality of a vertex v is n^2/t for some constant $t \geq 1$, then with probability $\geq 1 - 2\epsilon$ its centrality can be estimated to within a factor of $1/\epsilon$ with ϵt samples of source vertices.

The rest of this section is organized as follows. Please refer to the previous section for a review of the exact algorithm for betweenness centrality. We present our approximation algorithm based on adaptive sampling and its analysis in Section 3.4.1. Section 3.5.3 is an experimental study of our approximation technique on several real-world networks.

3.4.1 Adaptive-sampling based approximation

The adaptive sampling technique was introduced by Lipton and Naughton [130] for estimating the size of the transitive closure of a digraph. Prior to their work, algorithms for estimating transitive closure were based on randomly sampling source-vertices, solving the single-source reachability problem for the sampled vertices, and using this information to estimate the size of the transitive closure. The Lipton-Naughton algorithm introduces adaptive sampling of source-vertices, that is, the number of samples varies with the information obtained from each sample.

In this section, we give an *adaptive sampling* algorithm for computing betweenness of a given vertex v. It is a sampling algorithm in that it estimates the centrality by sampling a subset of vertices and performing SSSP computations from these vertices. It is termed *adaptive*, because the number of samples required varies with the information obtained from each sample.

The following lemma is easy to see.

Lemma 2. *$BC(v)$ is zero iff its neighboring vertices induce a clique.*

Let a_i denote the dependency of the vertex v_i on v i.e., $a_i = \delta_{v_i *}(v)$. Let $A = \sum a_i =$

$BC(v)$. It is easy to verify that $0 \leq a_i \leq n-2$ and $0 \leq A \leq (n-1)(n-2)/2$. The quantity we wish to estimate is A. Consider doing so with the following procedure:

> Repeatedly sample a vertex $v_i \in V$; perform SSSP (using BFS or Dijkstra's algorithm) from v_i and maintain a running sum S of the dependency scores $\delta_{v_i *}(v)$. Sample until S is greater than cn for some constant $c \geq 2$. Let the total number of samples be k. The estimated betweenness centrality score of v, $BC(v)$ is given by $\frac{nS}{k}$.

Let X_i be the random variable representing the dependency of a randomly sampled vertex on v. The probability of an event x is denoted by $\mathsf{Pr}[\, x \,]$. We establish the following lemmas to analyze the above algorithm.

Lemma 3. *Let $E[X_i]$ denote the expectation of X_i and $Var[X_i]$ denote the variance of X_i. Then, $E[X_i] = A/n$, $E[{X_i}^2] \leq A$, and $Var[X_i] \leq A$.*

The next lemma is useful in proving a lower bound on the expected number of samples before stopping.

Lemma 4. *5 Let $k = \epsilon n^2/A$. Then,*

$$\mathsf{Pr}[\, X_1 + X_2 + \cdots + X_k \geq cn \,] \leq \frac{\epsilon}{(c-\epsilon)^2}$$

Proof. We have

$$\begin{aligned}
\mathsf{Pr}[\, X_1 + \cdots + X_k \geq cn \,] &= \mathsf{Pr}\left[\left(X_1 - \frac{A}{n}\right) + \cdots + \left(X_k - \frac{A}{n}\right) \geq cn - \frac{kA}{n} \right] \\
&= \mathsf{Pr}\left[\left(X_1 - \frac{A}{n}\right) + \cdots + \left(X_k - \frac{A}{n}\right) \geq cn - \epsilon n \right] \\
&\leq \sum_i \mathsf{Pr}\left[X_i - \frac{A}{n} \geq (c-\epsilon)n \right] \\
&\leq \sum_i \frac{1}{(c-\epsilon)^2 n^2} Var[X_i] \\
&= \frac{1}{(c-\epsilon)^2 n^2} \sum_i Var[X_i] \\
&\leq \frac{1}{(c-\epsilon)^2 n^2} kA \\
&= \frac{\epsilon}{(c-\epsilon)^2}
\end{aligned}$$

Note that we have used Chebychev's inequality and union bounds in the above proof. We bound the error in the estimated value of A with the following lemma.

Lemma 5. *Let $k = \epsilon n^2/A$. Then,*

$$\mathsf{Pr}\,[\, X_1 + X_2 + \cdots + X_k \geq cn\,] \leq \frac{\epsilon}{(c-\epsilon)^2}$$

Lemma 6. *Let $k \geq \epsilon n^2/A$ and $d > 0$. Then*

$$\mathsf{Pr}\left[\, |\frac{n}{k}\left(\sum_{i=1}^{k} X_i\right) - A| \geq dA \,\right] \leq \frac{1}{\epsilon d^2}$$

Proof.

$$\begin{aligned}
\mathsf{Pr}\left[\, |\frac{n}{k}\left(\sum_{i=1}^{k} X_i\right) - A| \geq t \,\right] &= \mathsf{Pr}\left[\, |\left(\sum_{i=1}^{k} X_i\right) - \frac{k}{n}A| \geq \frac{kt}{n} \,\right] \\
&= \mathsf{Pr}\left[\, |\left(\sum_{i=1}^{k} X_i - \frac{1}{n}A\right)| \geq \frac{kt}{n} \,\right] \\
&\leq \frac{n^2}{k^2t^2} k \cdot Var[X_i]
\end{aligned}$$

Let $k = \lambda\dfrac{n^2}{A}$, where $\lambda \geq \epsilon$. Then the above probability is less than or equal to

$$\frac{n^2}{k^2t^2} k \cdot Var[X_i] \leq \frac{n^2}{\lambda\frac{n^2}{A}t^2} A$$

which is just $\dfrac{A^2}{\lambda t^2}$. Setting $Ad = t$ gives

$$\frac{1}{\lambda d^2} \leq \frac{1}{\epsilon d^2}$$

Theorem 2. *Let $\tilde{A}$ be the estimate of A in the above procedure and let $A > 0$. Then for $0 < \epsilon < 0.5$ with probability $\geq 1 - 2\epsilon$, the Algorithm in Section 3.4.1 estimates A to within a factor of $1/\epsilon$.*

Proof. There are two ways that the algorithm can fail: (i) it can stop too early to guarantee a good error bound, (ii) it can stop after enough samples but with a bad estimate.

First we claim that the procedure is unlikely to stop with $k \leq n^2/A$. We have that

$$\mathsf{Pr}\,[\,(\exists j)(j \leq k) \wedge (X_1 + X_2 + \cdots + X_j \geq cn)\,] \leq \mathsf{Pr}\,[\,X_1 + X_2 + \cdots + X_k \geq cn\,]$$

where $k = \dfrac{\epsilon n^2}{A}$, because the event to the right of the inequality implies the event to the left. But by Lemma 5, the right side of this equation is at most $\epsilon/(c-\epsilon)^2$. Substituting $c = 2$ and noting that $0 < \epsilon < 0.5$, we get that this probability is less than ϵ.

Next we turn to the accuracy of the estimate. If $k = \epsilon n^2/A$, by Lemma 6 the estimate,

$$\tilde{A} = \frac{n}{k}\sum_{i=1}^{k} X_i$$

is within dA of A with probability $\geq 1/(\epsilon d^2)$. Letting $d = 1/\epsilon$, this is just ϵ.

Putting the two ways of failure together, we get that the total probability of failure is less than $\epsilon + (1-\epsilon)\epsilon$, which is less than 2ϵ. Finally, note that if $A > 0$, there must be at least one i such that $a_i > 0$, so the algorithm will terminate. The case when $A = 0$ (i.e., centrality of v is 0) can be detected using Lemma 2 (before running the algorithm).

An interesting aspect of our theorem is that the sampling is adaptive. Usually such sampling procedures perform a fixed number of samples. Here it is critical that the algorithm adapts it behavior. Substituting $A = \dfrac{n^2}{t}$ in our analysis we get the following theorem.

Theorem 3. *For $0 < \epsilon < 0.5$, if the centrality of a vertex v is n^2/t for some constant $t \geq 1$, then with probability $\geq 1 - 2\epsilon$ its centrality can be estimated to within a factor of $1/\epsilon$ with ϵt samples of source vertices.*

Although our theoretical result is valid only for high centrality nodes, our experimental results (presented in the next section) show a similar behavior for all the vertices.

3.5 Approximate Betweenness Experimental Study

We assess the quality of the sampling-based approximation algorithm on several real-world graph instances (see Table 2). We use our parallel small-world graph analysis framework SNAP [17] to compute exact betweenness scores. Since the execution time and

speedup achieved by the approximation approach are directly proportional to the number of BFS/shortest path computations, we do not report performance results in this section. For a detailed discussion of exact centrality computation in parallel, and optimizations for small-world graphs, please refer to the previous chapter.

3.5.1 Problem Instances

Label	Network	n	m	Details	Source
`rand`	random graph	2000	7980	synthetic, undirected	[133]
`pref-attach`	preferential attachment	2000	7980	synthetic, undirected	[20]
`bio-pin`	human protein interactions	8503	32,191	undirected	[16]
`crawl`	web-crawl (`stanford.edu`)	9914	36,854	directed	[53]
`cite`	Lederberg citation network	8843	41,601	directed	[53, 26]
`road`	Rome, Italy road network	3353	4435	weighted, undirected	[57]

Table 2: Networks used in the approximate betweenness computation experimental study.

We experiment with two synthetic graph instances and four real networks in this study. `rand` is an unweighted, undirected random network of 2000 vertices and 7980 edges, generated using the Erdős–Rényi graph model [68]. This synthetic graph has a low diameter, low clustering, and a Gaussian degree distribution. `pref-attach` is a synthetic graph generated using the Preferential attachment model proposed by Barabási and Albert [21]. This model generates graphs with heavy-tailed degree distributions and scale-free properties. Vertices are added one at a time, and for each of them, we create a fixed number of edges connecting to existing vertices, with probability proportional to their degree. `bio-pin` is a biological network that represents interactions in the human proteome [154, 16]. This graph is undirected, unweighted and exhibits small-world characteristics. `crawl` corresponds to the `wb-cs-stanford` network in the UF sparse matrix collection [53]. It is a directed graph, where vertices correspond to pages in the Stanford Computer Science domain, and edges represent links. `cite` is a directed graph from the Pajek network collection [26]. It corresponds to papers by and citing J. Lederberg (1945-2002). `road` is a weighted graph of 3353 vertices and 4435 edges that corresponds to a large portion of the road network of Rome, Italy from 1999 [57]. Vertices correspond to intersections between roads, and edges correspond to roads or road segments. Edge weights are physical distances in metres. Road networks have more structure and a higher diameter than the other networks considered in

this study.

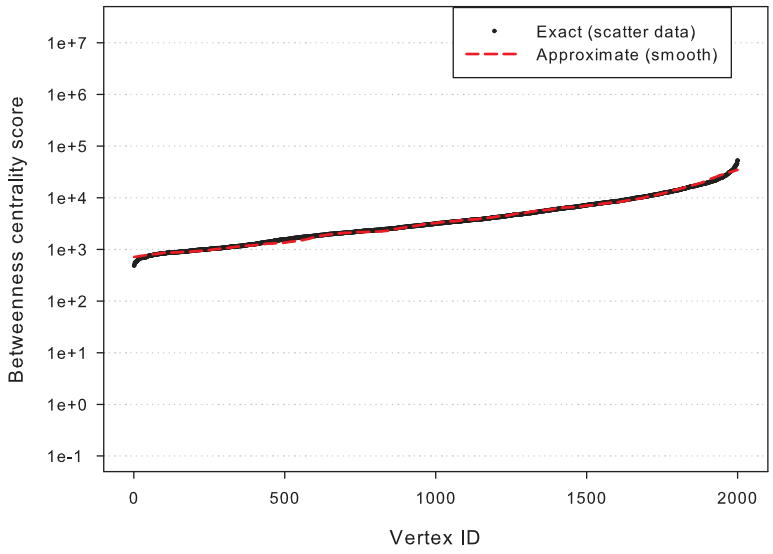
3.5.2 Methodology

Our goal in this study is to quantify the approximation quality, and so we primarily compare the approximation results to exact scores. We first compute exact centrality scores of all the networks in Table 2. In most data sets, we are interested in high-centrality vertices, as they are the critical entities and are used in further analysis. From the exact scores, we identify vertices whose centrality scores are an order of magnitude greater than the rest of the network. For these vertices, we study the trade-off between computation and approximation quality by varying the parameter c in the centrality estimation algorithm. We also show that it is easy to estimate scores of low-centrality vertices. We chose small networks for ease of analysis and visualization, but the approximation algorithm can be effectively applied to large networks as well (see, for instance, the networks considered in [15]).

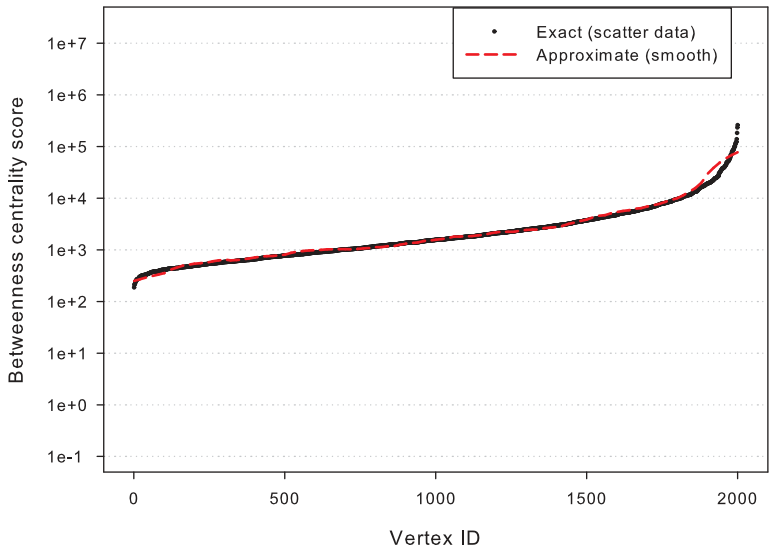
3.5.3 Results and Analysis

Figures 26, 27 and 28 plot the distribution of exact and approximate betweenness scores for the six different test instances. The approximate betweenness scatter data is smoothed by a local smoothing technique using polynomial regression. Note that the synthetic networks, `rand` and `pref-attach` show significantly lower variation in exact centrality scores compared to the real instances. Also, there are a significant percentage of low-centrality vertices (scores less than, or close to, n) in `cite`, `crawl` and `bio-pin`.

We apply the approximate betweenness algorithm to estimate betweenness centrality scores of all the vertices in the test instances. In order to visualize the data better, we plot a smoothed curve of the estimated betweenness centrality data that is superimposed with the exact centrality score scatter-plot. We set the parameter c in the algorithm described in Section 3.4.1 to 5 for these experiments. In addition, we impose a cut-off of $\frac{n}{20}$ on the number of samples. Observe that in all the networks, the estimated centrality scores are very close to the exact ones, and we are guaranteed to cut down on the computation by a factor of nearly 20.

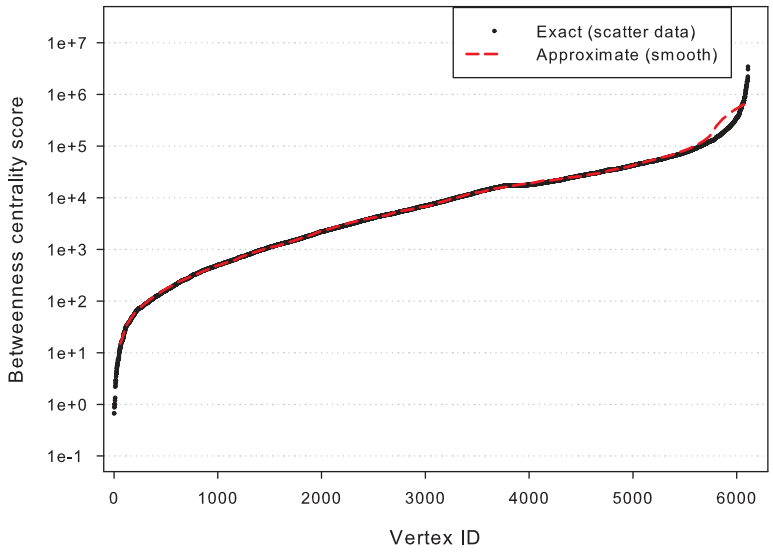


(a) `rand`

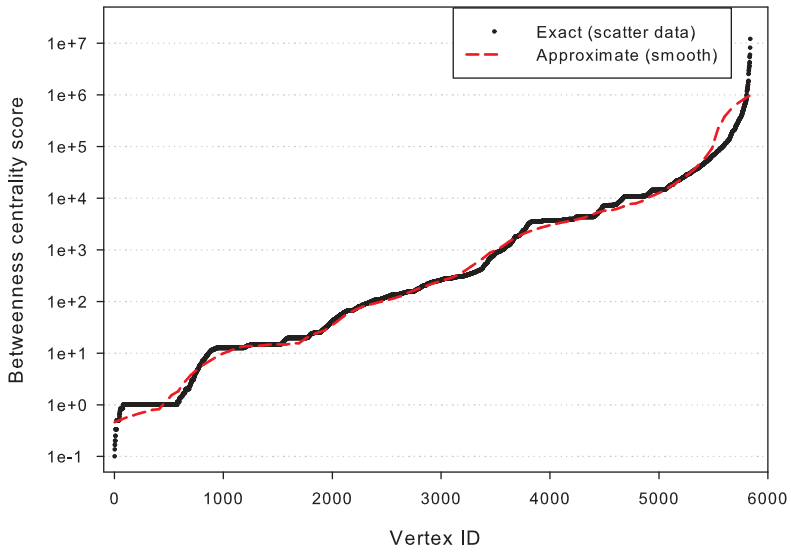


(b) `pref-attach`

Figure 26: A scatter plot of exact betweenness scores of all the vertices (in sorted order) in `rand` and `pref-attach`, and a line plot of their estimated betweenness scores.

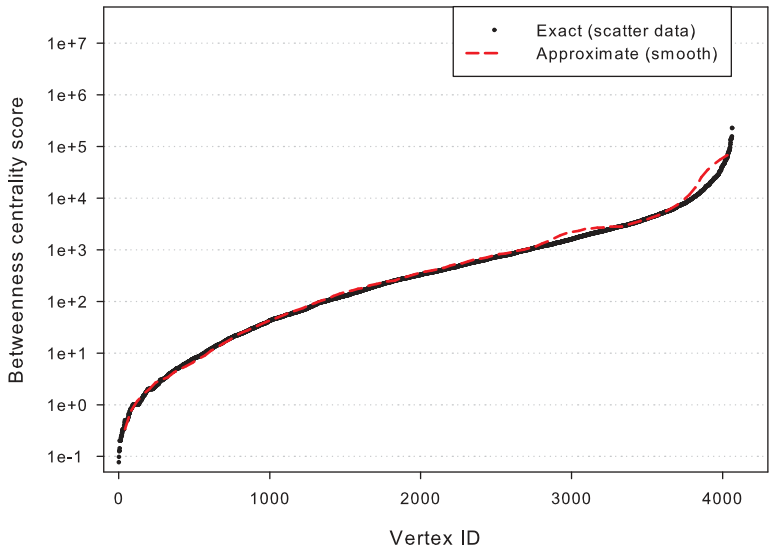


(a) `bio-pin`

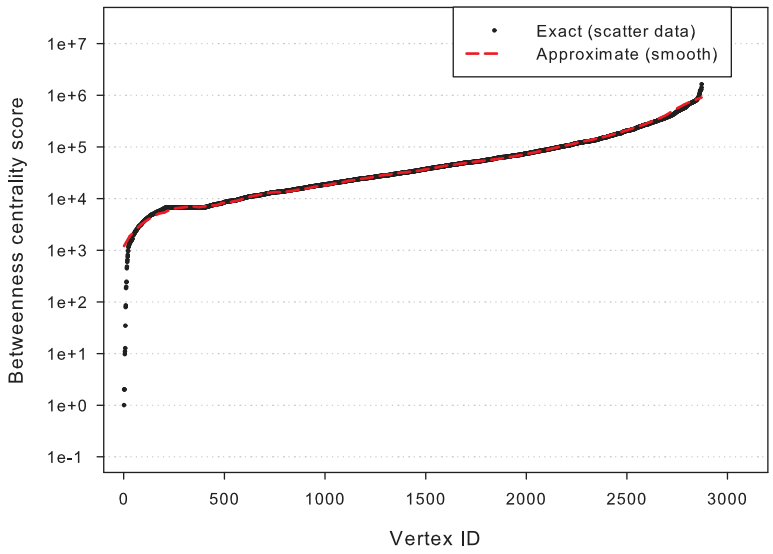


(b) `pref-attach`

Figure 27: A scatter plot of exact betweenness scores of all the vertices (in sorted order) in `bio-pin` and `crawl`, and a line plot of their estimated betweenness scores.



(a) `cite`



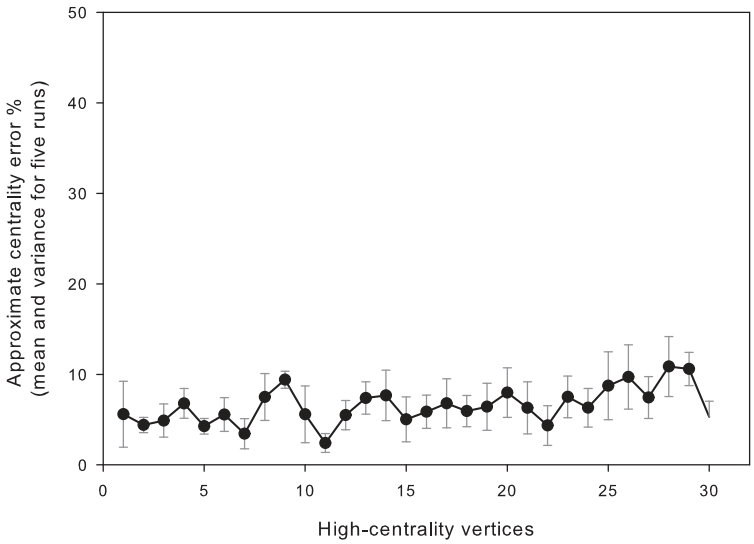
(b) `road`

Figure 28: A scatter plot of exact betweenness scores of all the vertices (in sorted order) in `cite` and `road`, and a line plot of their estimated betweenness scores.

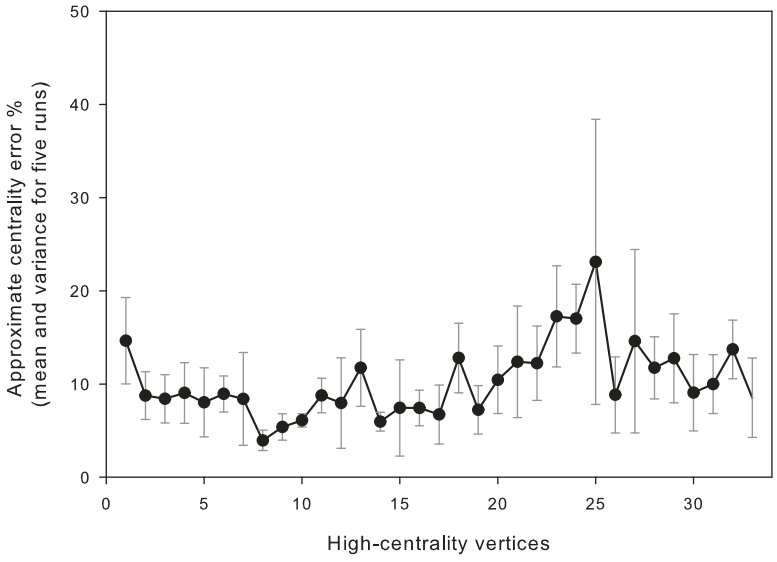
To further study the quality of approximation for high-centrality vertices, we select the top 1% of the vertices (about 30) ordered by exact centrality score in each network, and compute their estimated centrality scores using the adaptive-sampling algorithm. Since the source vertices in the adaptive approach are chosen randomly, we repeat the experiment five times for each vertex and report the mean and variance in approximation error. Figures 29, 30, and 31 plot the mean percentage approximation error in the computed scores for these high centrality vertices, when the value of c (see Algorithm 3) is set to 5. The vertices are sorted by exact centrality score on the X-axis. The error bars in the charts indicate the variance in estimated score due to random runs, for each network. For the random graph instance, the average error is about 5%, while it is roughly around 10% for the rest of the networks. Except for a few anomalous vertices, the error variance is within reasonable bounds in all the graph classes.

Figures 32, 33, and 34 plot the percentage of BFS/SSSP computations required for approximating the centrality scores, when c is set to 5. This algorithmic count is an indicator of the amount of work done by the approximation algorithm. The vertices are ordered again by their exact centrality scores from left to right, with the vertex with the least score to the left. A common trend we observe across all graph classes is that the percentage of source vertices decreases as the centrality score increases – this implies that the scores of high centrality vertices can be approximated with lesser work using the adaptive sampling approach. Also, this value is significantly lower for `crawl`, `bio-pin` and `road` compared to other graph classes.

We can also vary the parameter c, which affects both the percentage of BFS/SSSP computations and the approximation quality. Table 3 summarizes the average performance on each graph instance, for different values of c. Taking only high-centrality vertices into consideration, we report the mean approximation error and the number of samples for each graph instance. As expected, we find that the error decreases as the parameter c is increased, while the number of samples increases. Since the highest centrality value is around $10 * n$ for the citation network, a significant number of shortest path computations have to be done even for calculating scores with a reasonable accuracy. But for other graph instances,

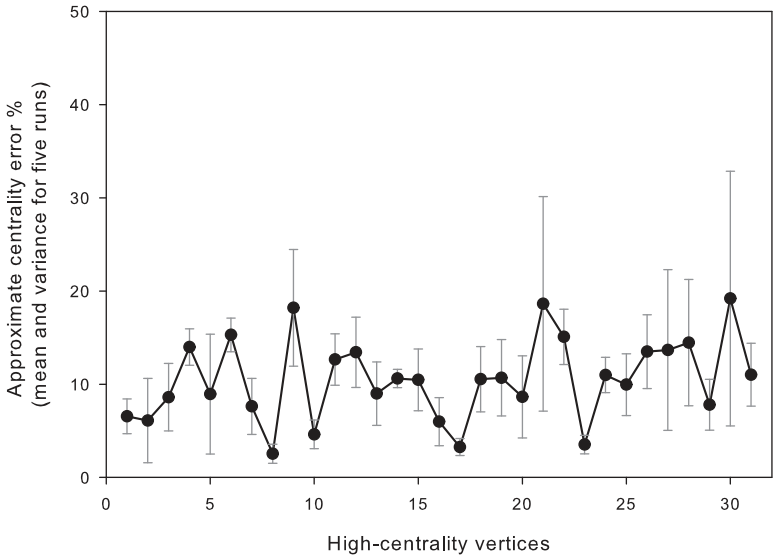


(a) `rand`

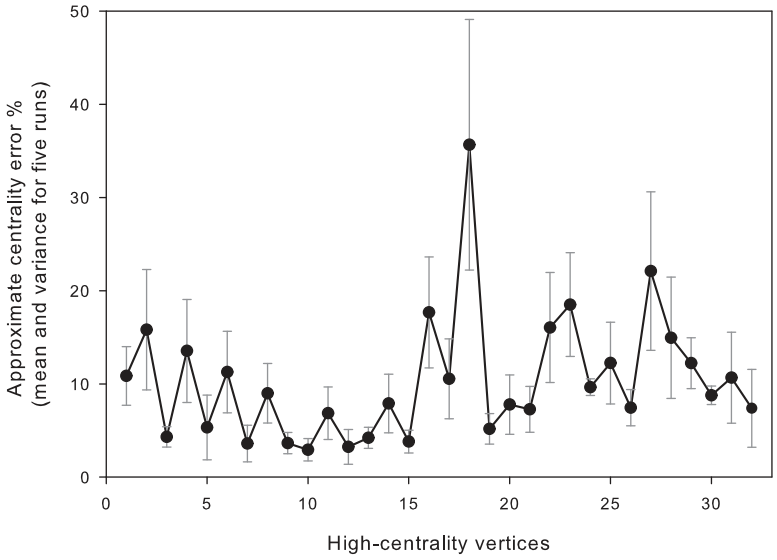


(b) `pref-attach`

Figure 29: Average estimated betweenness error percentage (in comparison to the exact centrality score) for multiple runs (`rand` and `pref-attach` networks). The adaptive sampling parameter c is set to 5 for all experiments and the error bars indicate the variance.

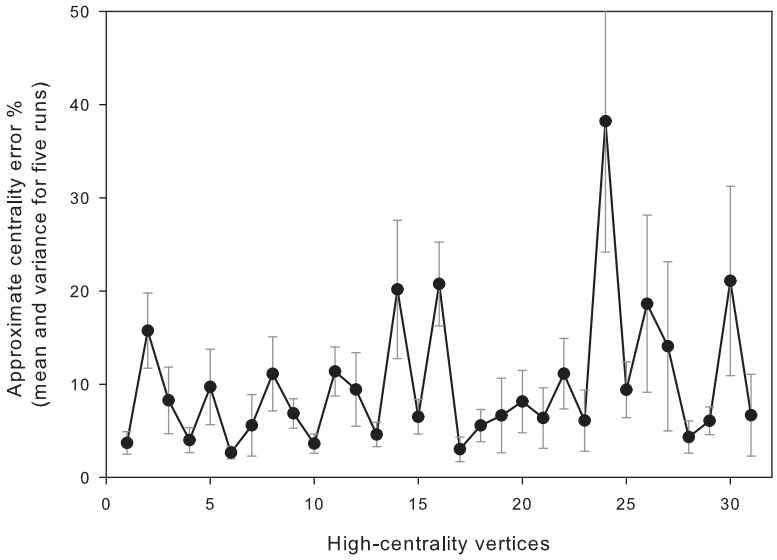


(a) `bio-pin`

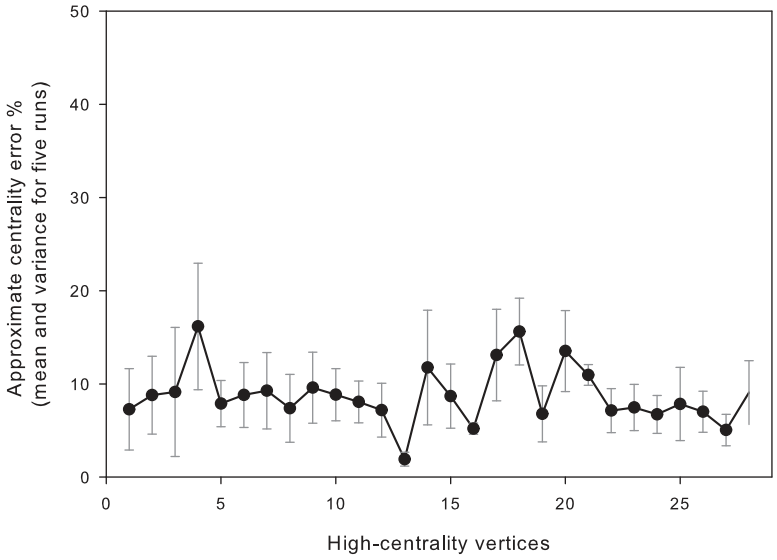


(b) `crawl`

Figure 30: Average estimated betweenness error percentage (in comparison to the exact centrality score) for multiple runs (`bio-pin` and `crawl` networks). The adaptive sampling parameter c is set to 5 for all experiments and the error bars indicate the variance.

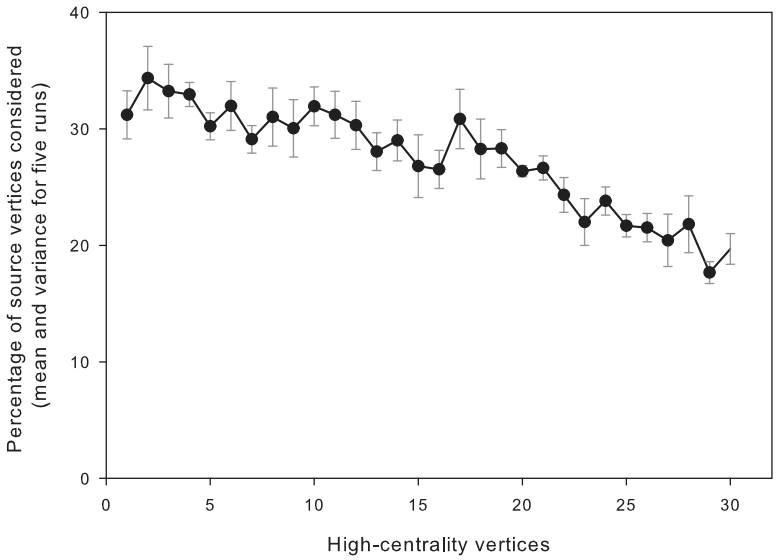


(a) `cite`

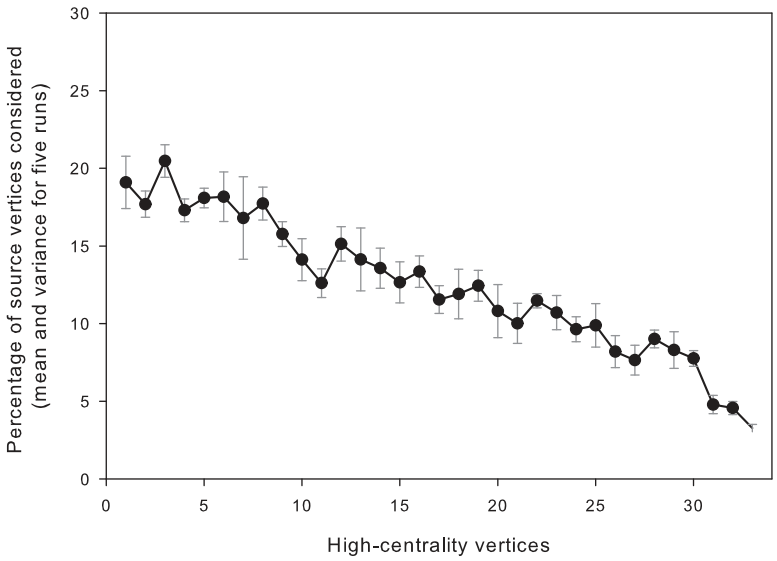


(b) `road`

Figure 31: Average estimated betweenness error percentage (in comparison to the exact centrality score) for multiple runs (`cite` and `road` networks). The adaptive sampling parameter c is set to 5 for all experiments and the error bars indicate the variance.

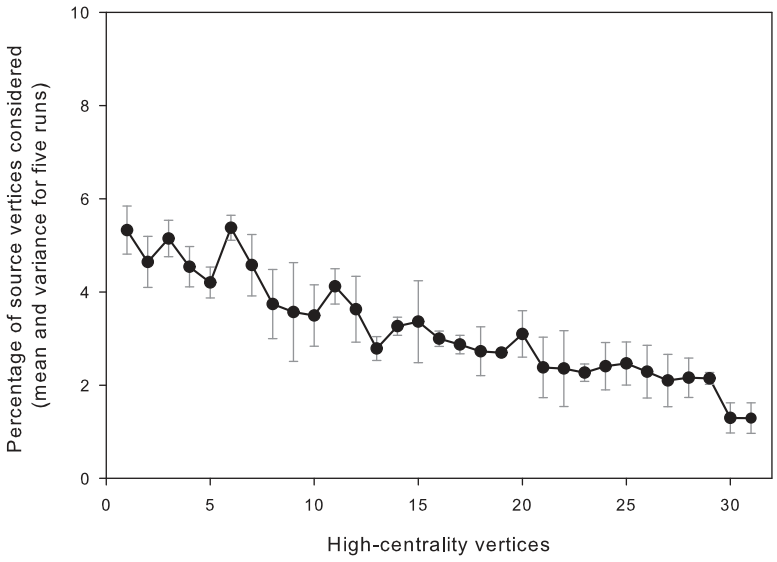


(a) `rand`

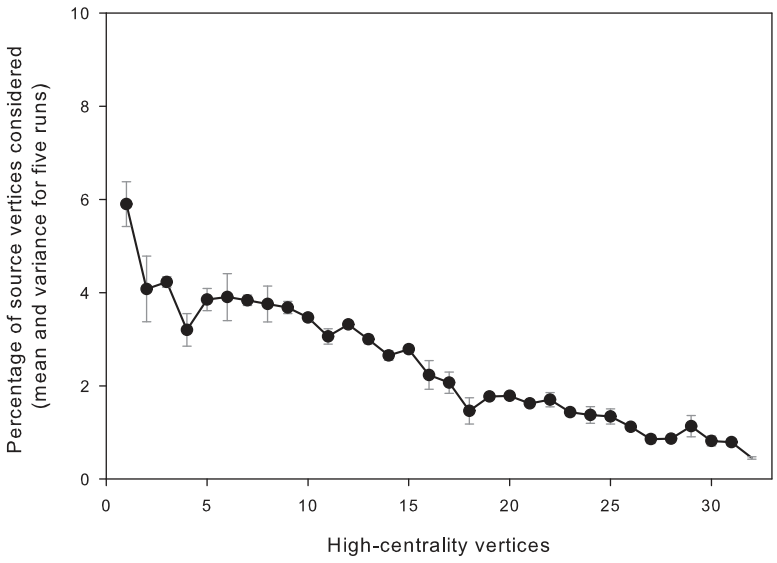


(b) `pref-attach`

Figure 32: The number of samples/SSSP computations as a fraction of n, the total number of vertices (`rand` and `pref-attach` networks). This algorithmic count is an indicator of the amount of work done by the approximation algorithm. The adaptive sampling parameter c is set to 5, and the error bars indicate the variance from 5 runs.

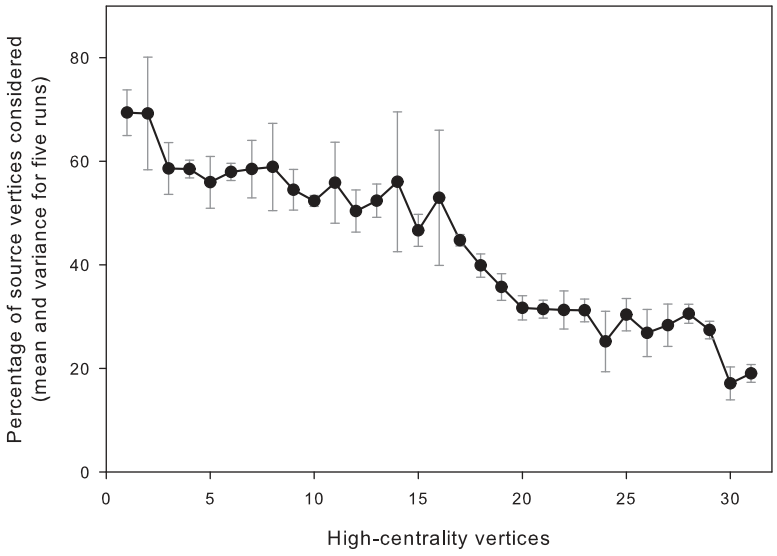


(a) `bio-pin`

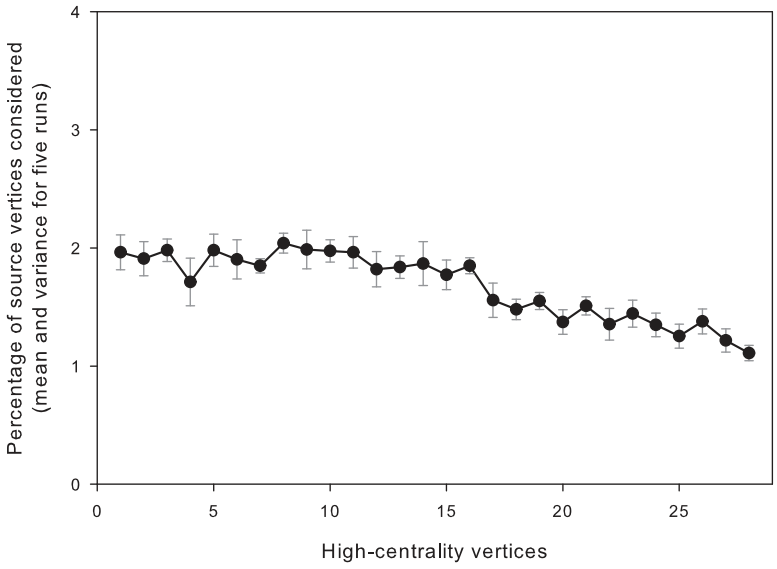


(b) `crawl`

Figure 33: The number of samples/SSSP computations as a fraction of n, the total number of vertices (`bio-pin` and `crawl` networks). This algorithmic count is an indicator of the amount of work done by the approximation algorithm. The adaptive sampling parameter c is set to 5, and the error bars indicate the variance from 5 runs.



(a) `cite`



(b) `road`

Figure 34: The number of samples/SSSP computations as a fraction of n, the total number of vertices (`cite` and `road` networks). This algorithmic count is an indicator of the amount of work done by the approximation algorithm. The adaptive sampling parameter c is set to 5, and the error bars indicate the variance from 5 runs.

particularly the road network, web-crawl and the protein interaction network, $c = 5$ offers a good trade-off between computation and the approximation quality.

Network	`rand`	`pref-attach`	`bio-pin`	`crawl`	`cite`	`road`
t = 2						
Avg. error	16.28%	29.39%	46.72%	33.69%	32.51%	22.58%
Avg. $\frac{k}{n}$	11.31%	5.36%	1.30%	0.96%	17.00%	0.68 %
t = 5						
Avg. error	6.51%	10.28%	10.49%	10.31%	9.98%	8.79%
Avg. $\frac{k}{n}$	27.37%	12.38%	3.20%	2.42%	43.85%	1.68%
t = 10						
Avg. error	5.62%	6.13%	7.17%	7.04%	–	7.39%
Avg. $\frac{k}{n}$	54.51%	24.66%	6.33%	4.89%	–	3.29%

Table 3: Approximate betweenness computation: Observed average-case algorithmic counts, as the value of the sampling parameter c is varied. The average error percentage is the deviation of the estimated score from the exact score, and the $\frac{k}{n}$ percentage indicates the number of samples/SSSP computations.

3.6 Centrality case study: Human protein interactome analysis

In this Section, we demonstrate the application of high performance computing techniques in Systems biology and present multicore algorithms for the important research problem of protein-interaction network (PIN) analysis. PINs play an important role in understanding the functional and organizational principles of biological processes. Promising computational techniques for key systems biology research problems such as identification of signaling pathways, novel protein function prediction, and the study of disease mechanisms, are based on topological characteristics of the protein interactome. Several complex network models have been proposed to explain the evolution of protein networks, and these models primarily try to reproduce the topological features observed in yeast, the model eukaryote interactome. We study the structural properties of a high-confidence *human* interaction network, constructed by assimilating recent experimentally derived interaction data. We identify topological properties common to the yeast and human protein networks.

A novel contribution of our work is the analysis of the degree-betweenness centrality correlation in the human PIN. Jeong et al. empirically showed that betweenness is positively

correlated with the essentiality and evolutionary age of a protein. We observe that proteins with high betweenness, but low degree (or connectivity) are abundant in the human PIN. Our parallel implementations for the exact calculation of betweenness and other compute-intensive centrality metrics can also be applied to interactome analysis. For instance, on the Sun Fire T2000 server with the UltraSparc T1 (Niagara) processor, we achieve a relative speedup of about 16 using 32 threads for a typical instance of betweenness centrality on a PIN, reducing the running time from nearly $3\frac{1}{2}$ minutes to 13 seconds.

3.6.1 Protein Interaction Networks

Recent advances in high-throughput genomic experimental techniques have resulted in an abundance of diverse gene sequence and structure data. As a consequence, we are also faced with a significant volume of novel, unannotated gene products. The traditional methods of gene and protein annotation, such as homology-based transfer, are insufficient to characterize novel proteins, and are proving to be erroneous in many cases. This has led to a shift in research focus from the study of individual proteins to an integrative analysis of global characteristics and interactions between various cellular components using quantitative approaches. This research field, Systems biology, has served as the foundation for the reconstruction of metabolic pathways, regulatory and signaling networks, and the identification of disease mechanisms. Protein function prediction is one of the key drivers for Systems biology research. There are various approaches available for deducing the function of novel proteins, among which the study of interaction networks is one of the most promising techniques [176, 30, 179].

In order to design efficient computational techniques that are based on global connectivity patterns, it is essential to understand the topology of the network first. Genomic research in the past few years has enabled us to map high-confidence interactomes of model eukaryotes such as yeast [176, 174], worm [128] and fly [81]. These protein-interactions are mainly derived using the yeast two-hybrid (Y2H) assay technique, and have provided encouraging evidence that global topological structure and network features relate to known biological properties [108]. This has in-turn motivated several research groups to work on

a global map of the *human interaction network*, and there have been several recent efforts on mapping the global human genome [162] using the Y2H assay. However, this system is prone to a high rate of false-positives and the interactions need to be validated with sophisticated techniques. Also, the identity of essential interactions in PINs differ significantly, depending on the experimental methodology [24]. The high-confidence interactions have been identified, filtered and are now readily available from online public domain databases (for example, BIND [5], DIP [164] and HPRD [154]). Most of these databases are literature-based and hand-curated with a sizable percentage of overlapping interactions.

The interaction networks of model eukaryotes such as yeast are analyzed extensively [185, 110] using graph-theoretic and complex network analysis concepts. The yeast protein interaction network (PIN) topology exhibits several interesting features that distinguish it from a random graph. For instance, the yeast PIN contains a larger number of highly-connected (high-degree) proteins than one would expect in a random Erdős-Rényi network. Jeong et al. also observed that in the yeast network, the connectivity of a protein appears to be positively correlated with its essentiality [108], i.e., highly connected proteins tend to be more essential to the viability of the organism. Joy et al. [110] report that in the yeast network, proteins with high betweenness are more likely to be essential, and that the evolutionary age of proteins is positively correlated with betweenness. Also, they observe that there are several proteins with low degree but high centrality scores in the yeast PIN.

Gandhi et al. [78] present a comprehensive analysis of a large-scale human interaction network [126, 159]. They study a dataset of about 26,000 human protein interactions obtained from various public databases, compare the human interactome with the yeast, worm and fly datasets, and observe that only 42 interactions were common to all species. Also, they observe that unlike the yeast network, the available human PIN data does not support the presumption on the positive correlation between connectivity and essentiality.

We extend the work of Gandhi et al. [78] and Joy et al. [110] in this article. Our main contributions are the following:

- *Topological study of the largest human PIN constructed to date, comprising nearly 18,000 proteins and 34,000 interactions.* We analyze the global connectivity and

clustering properties of a human PIN composed of high-confidence pair-wise protein interactions. We do not model complex interactions as pair-wise interactions, since it is not always known which proteins in the complex interact with each other.

- *Computation of centrality metrics for the human PIN.* We analyze betweenness centrality scores and find that proteins with high betweenness centrality but low connectivity are abundant in the human PIN. We also observe that this finding cannot be explained by the widely-accepted models for scale-free networks.

- *Applying high performance computing techniques for large-scale PIN analysis.* Our efficient multicore implementation reduces the computation time of betweenness centrality to 13 seconds on 32 processors of the Sun Fire T2000 system, with a relative speedup of 16.

3.6.2 Interactome datasets

There are several online databases devoted to the human interactome (see Table 4). In our previous study of the human PIN [16], we constructed the interaction map by merging information from Gandhi et al.'s human proteome analysis dataset [78] (updated February 2006), an interaction dataset from the Human Protein Reference Database [154] (updated May 2006), and IntAct (updated October 2006). The interaction network used in this article (referred to as HPIN throughout this article) is an updated dataset (January 2007) of binary interactions from HPRD. The latest version of the HPRD dataset includes interactions from MIPS, BIND, DIP and MINT. There is a complication using protein complex data (for example, from the MIPS database) to obtain protein interactions, since it is not always known which proteins in a complex interact with each other. Note that we *do not model* complex interactions as pair-wise interactions in this study.

We also present the topological characteristics of two large-scale yeast PINs for comparison with HPIN. The yeast PIN from Jeong et al. [108] (YPIN) is an undirected network of 2112 proteins and 7182 interactions, while Reguly et al. [160] (YPIN2) provide an undirected network of 3289 proteins and 11334 interactions. Both the network are well-studied, and all the reported interactions are high-confidence ones.

Table 4: Online human protein interaction databases.

Database	Details
HPRD [154]	Human Protein Reference Database. Experimentally verified protein-protein interactions obtained from manual curation of literature. 25205 proteins and 37581 interactions.
BIND	Biomolecular Interaction Network Database. Collection of molecular interactions including high-throughput data submissions and hand-curated information from the scientific literature. 4644 human protein interactions.
MIPS	Munich Information Center for Protein Sequences. 334 interactions.
MINT	Molecular Interactions Database. 3544 interactions.
IntAct [98]	Freely available, open source database system and analysis tools for protein interaction data. European Bioinformatics Institute. 2420 interactions.
OPHID	Online Predicted Human Interaction Database. Repository of already known experimentally derived human protein interactions, as well as 23,889 additional predicted interactions. This dataset is not included in our human PIN.

3.6.3 Parallel Multi-core Performance

The sequential complexity for computing betweenness centrality and other shortest-path based centrality metrics is $O(mn)$. The parallel algorithms for betweenness centrality described in the prior section are well-suited for implementation on multicore and multiprocessor systems that have high memory bandwidth and a modest number of processor cores. While betweenness is compute-intensive, finding the clustering coefficients, assortativity, the joint degree distribution, and other topological measures are straight-forward to compute with linear-work algorithms. Portable, efficient implementations of these algorithms are freely available from our website as part of the SNAP (Small-world Network Analysis and Partitioning) framework [17]. As a representative case study for performance on multicore systems, we will present results of computing betweenness, closeness centrality, and diameter for HPIN in this section.

We report performance results on the Sun Fire T2000 multi-core server, with the Sun UltraSPARC T1 (Niagara) processor. This system has eight cores running at 1.0 GHz, each of which is four-way multithreaded. There are eight integer units with a six-stage pipeline on chip, and four threads running on a core share the pipeline. The cores also share a 3

MB L2 cache, and the system has a main memory of 16 GB. Since there is only one floating point unit (FPU) for all cores, the UltraSparc T1 processor is mainly suited for programs with few or no floating point operations. We compile our codes with the Sun C compiler v2.8 and the flags `-xtarget=ultraT1 -xarch=v9b -xopenmp`.

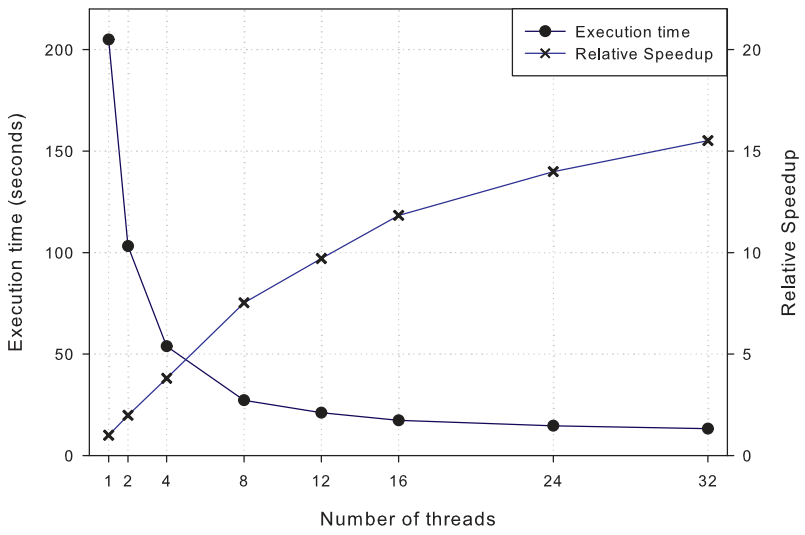


Figure 35: Sun Fire T2000 parallel performance for betweenness centrality computation on HPIN.

Figure 35 plots the execution time and relative speedup achieved on the Sun Fire T2000 for the exact computation of betweenness centrality. The performance scales nearly linearly up to 16 threads, but drops between 16 and 32 threads. This can be attributed to insufficient memory bandwidth on 32 threads, as well as the presence of only one floating point unit on the entire chip. We use the floating point unit for accumulating pair dependencies and centrality values. The execution times for other shortest path-based centrality metrics such as closeness (Figure 36) and betweenness centrality differ by a constant multiplicative factor. Betweenness centrality computation is much more involved, as it requires maintaining a BFS stack, a queue and a predecessor list. Also, the BFS tree is traversed twice in the algorithm. For additional performance results and a discussion of scalability related to network properties, please refer to [12, 15].

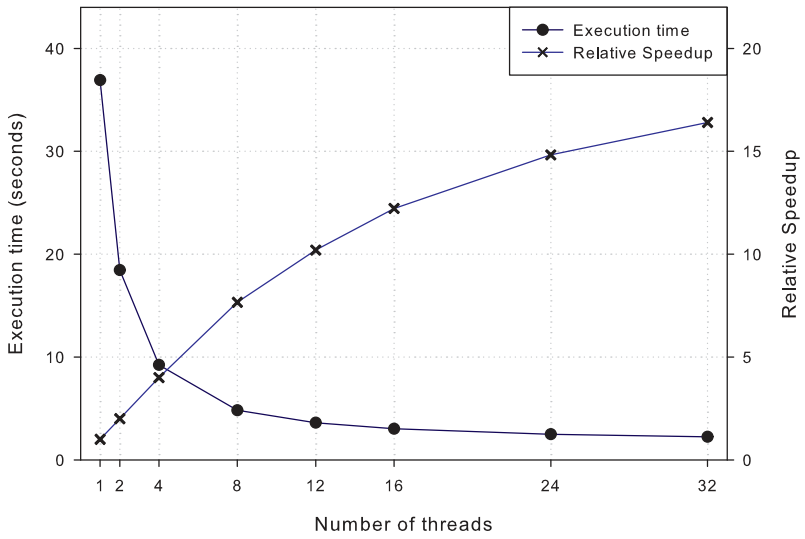


Figure 36: Sun Fire T2000 parallel performance for closeness centrality computation on HPIN.

Computing the graph diameter and average path length also have a $O(mn)$ work complexity, and the parallel approach is very similar to betweenness centrality. In both cases, we need to run n Breadth-First searches. We report performance results for computing the graph diameter on a dual-core 2.8 GHz Intel Xeon system. The two cores share a 2 MB L2 cache, and the system has 4 GB physical memory. Each processor is also equipped with hyper-threading, which gives an impression of four virtual processors as a whole. The code is compiled with the Intel C Compiler v9.1 and the flags (`-O3 -ipo -unroll_loops`). Fig 37 plots the execution time and speedup as the number of threads is varied from 1 to 4. We achieve near-linear speedup up to 4 threads in this case.

3.6.4 Biological Analysis

In this section, we will quantify *connectivity*, *centrality* and *clustering* in the human and yeast PINs using relevant social network analysis metrics.

The HPIN dataset we obtain from HPRD [154] has 18755 proteins and 34367 pair-wise interactions. We process this network and extract the largest connected component, which is composed of 8503 proteins and 32191 interactions. Thus, the original dataset includes around 10000 non-interacting proteins. Apart from the large component, there are 89

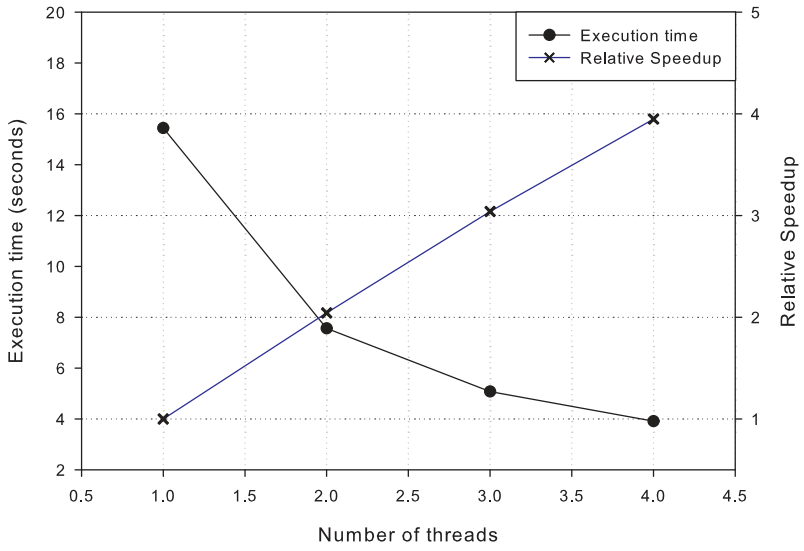


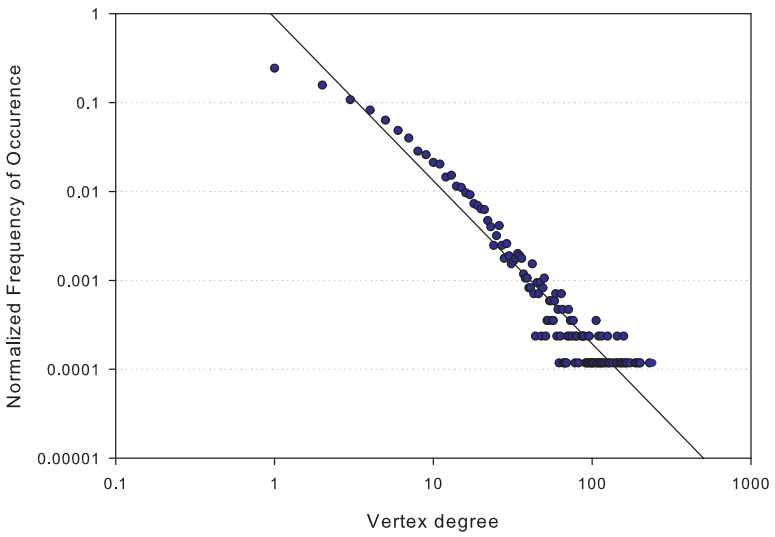
Figure 37: Execution time and speedup on a dual-core Intel Xeon system for graph diameter computation.

connected components of size 2, 16 connected components of size 3 and 5 components of size 5. We also ignore complex interactions in the dataset.

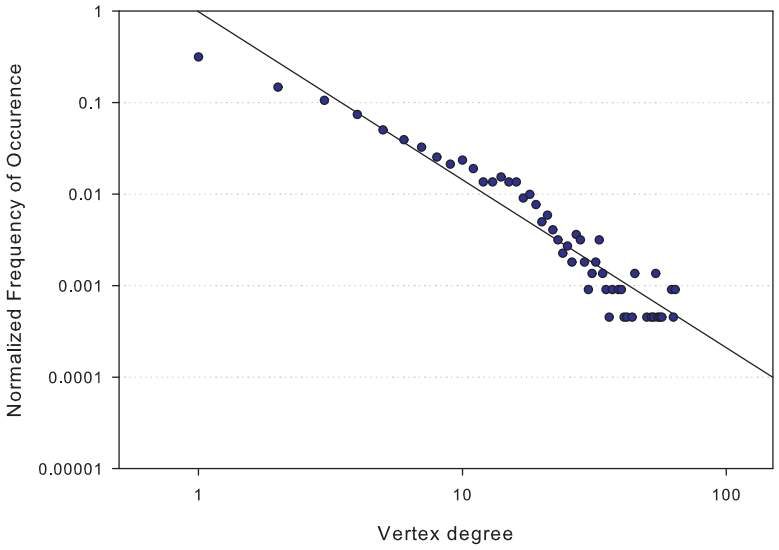
Connectivity. Figure 38(a) plots the degree distribution of the vertices in the largest component of HPIN. We report the normalized frequency count $p(k) = \frac{n(k)}{n}$, where $n(k)$ is the total number of degree-k vertices. Similarly, Figure 38(b) gives the degree distribution of the yeast PIN. In HPIN, nearly 25% of vertices have a degree of 1, and 15% are of degree 2. In comparison, 31% of vertices are of degree 1 and 15% are degree 2 in YPIN. The degree distribution can be roughly approximated by a power-law, but with a heavy tail.

The protein with the highest degree in HPIN is *TP53* (230 interactions). TP53 stands for tumor-protein 53 – it regulates the cell division process by keeping cells from growing and dividing too fast, or in an uncontrolled way. The p53 tumor protein is located in the nucleus of cells throughout the body and can bind directly to DNA. Since the p53 tumor protein is essential for regulating cell division, it is also known as the *guardian of the genome*.

Clustering. We calculate the average clustering coefficient, a measure of the tendency of proteins in a network to form clusters or groups. For a vertex v of degree d, the clustering coefficient CC is defined as the $CC(v) = \frac{2k}{d(d-1)}$, where k is the number of links connecting



(a) Human PIN



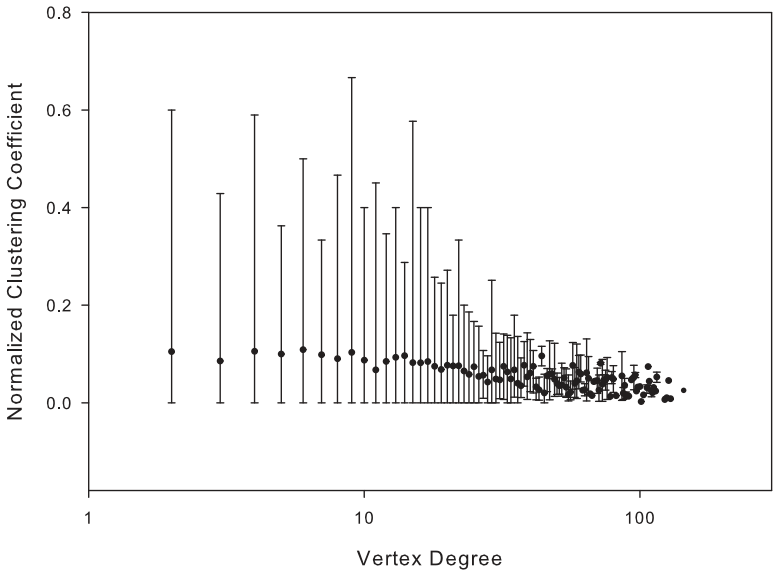
(b) Yeast PIN

Figure 38: Vertex degree distributions of the human and yeast protein interaction networks.

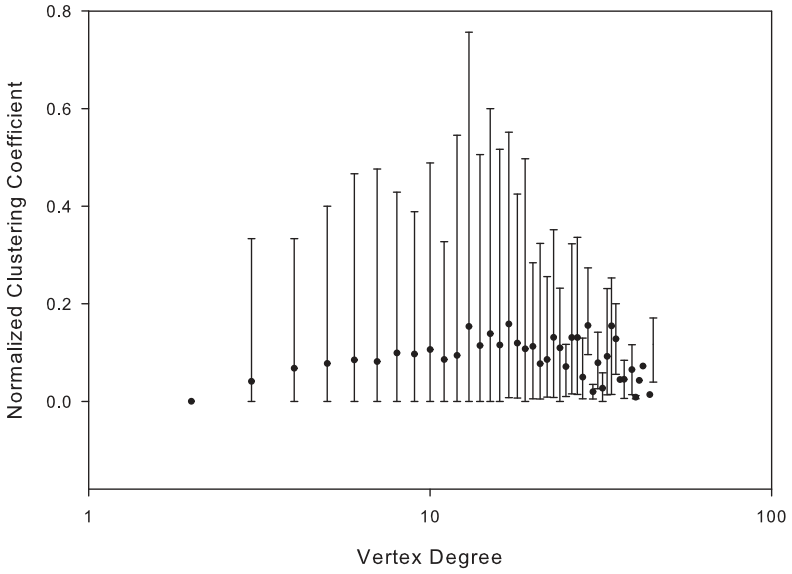
the d neighbors of v, considered pairwise. The average clustering coefficient $CC_a(d)$ for a particular degree d is simply the average of the clustering coefficients of all vertices of degree d. We find that, on an average, CC_a is a constant value of 0.1 for both HPIN and YPIN (see Figure 39). This is a strong indicator that these networks *do not* have a hierarchical organization, or trivially noticeable community structure [82]. Newman [145] observes that networks with high clustering coefficients are prone to virus outbreaks, and faster epidemic spreading. A constant average clustering coefficient across the network might be one of the distinguishing features of PINs. Also, we observe that low-degree vertices in HPIN and YPIN show some variation in the clustering coefficient values, whereas high-degree vertices (degree greater than 20) show little or no variation (see error bars in Figure 39). Also, note that we do not model protein complexes as pairwise interactions in this study. Complex interactions lead to an occurrence of proteins of sizable degree as well as high clustering-coefficients in the graph.

There are several metrics to study correlations between vertex degree and the connectivity of neighbors of that vertex. Based on extensive empirical studies, Newman [145] proposes a simple classification of networks into three classes: assortative, disassortative, and neutral mixing [145]. A vertex with high degree in an assortative (disassortative) network tends to connect to nodes with other higher (low) degree vertices, whereas in a neutral mixing there are no such patterns. Disassortative networks are vulnerable to both random failures and targeted attacks at the high-degree vertices. Other metrics, such as likelihood, radial, and tangential, are also directly related to assortativity. Tangential links are used to refer to edges connecting vertices of similar degrees, and radial links refer the links connecting high degree vertices with low degree ones. On calculating Newman's assortativity coefficient, we found that both HPIN and YPIN exhibit mildly disassortative to neutral mixing.

The joint degree distribution (JDD) is another metric similar to assortativity that is used to study clustering. JDD is the probability that a randomly selected edge connects vertices of degree k_1 and k_2 respectively, $P(k_1, k_2) = \frac{m(k_1,k_2)}{m}$, where $m(k_1, k_2)$ is the total number of edges between vertices of degree k_1 and k_2 respectively. We plot the joint degree distributions of HPIN and YPIN in Figure 40. In HPIN, we find that the majority of edges



(a) Human PIN



(b) Yeast PIN

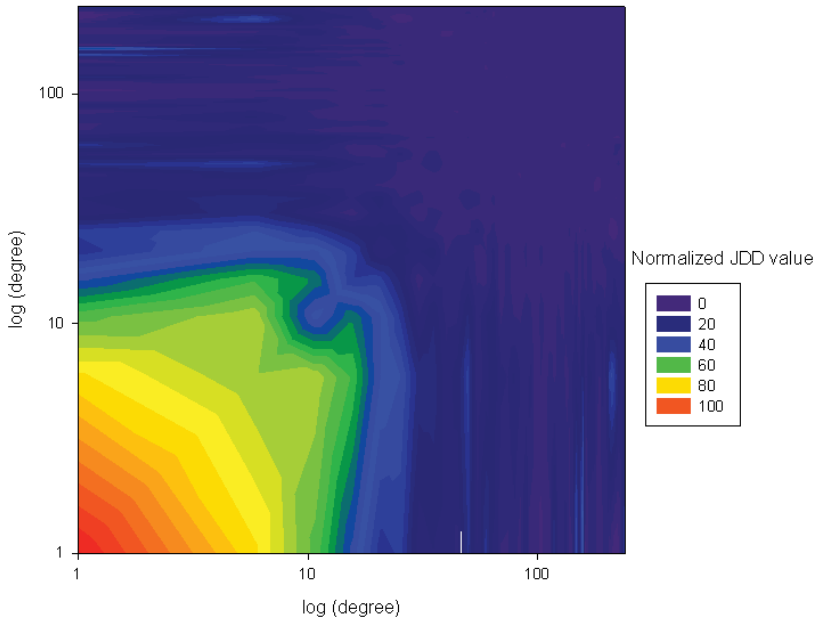
Figure 39: Average clustering coefficient for degree-k vertices (the error bars indicate the maximum and minimum values).

are links between low-degree vertices (bottom-left corner). For yeast, there are some links connecting high-degree vertices with low-degree ones (bottom-right and top-left). However, observe that there are no tangential links between high-degree vertices (top-right). This is an important network characteristic for which the PIN topology differs significantly from a physical network, such as the AS-level network topology.

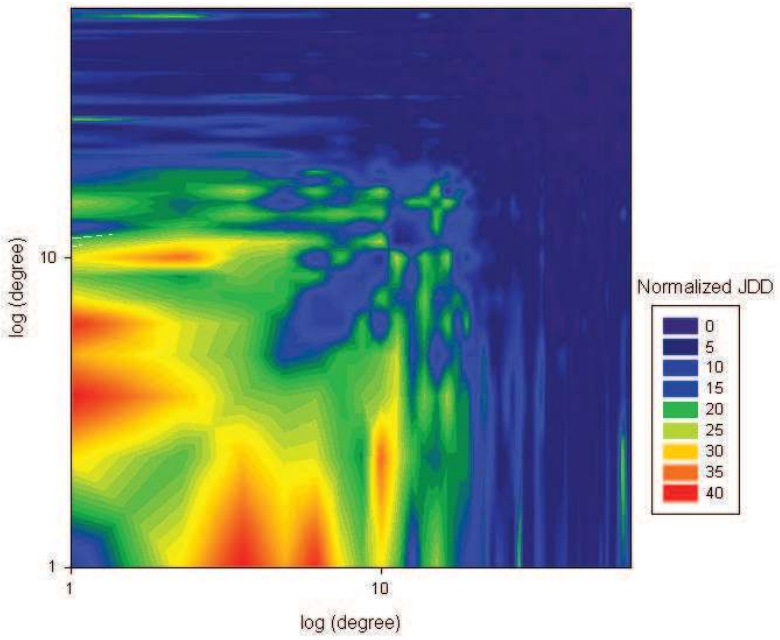
Centrality. Next we study centrality and criticality in PINs. We will use the Betweenness centrality metric to analyze the human interactome. Researchers have paid particular attention to the relation between centrality and *essentiality* or *lethality* of a protein (for instance, [108]). A protein is said to be essential if the organism cannot survive without it. Essential proteins can only be determined experimentally, so alternate approaches to *predicting essentiality* are of great interest and have potentially significant applications such as drug target identification [109]. Previous studies on yeast have shown that proteins acting as hubs (or high-degree vertices) are three times more likely to be essential [108]. So we wish to analyze the interplay between degree and centrality scores for proteins in the human PIN in this section.

Figure 41 plots the normalized betweenness centrality scores (absolute centrality score divided by $(n-1)*(n-2)/2$, the highest possible centrality score for a vertex in an undirected network) of all the proteins in HPIN, ordered by degree. We repeat the analysis for both the yeast datasets (see Figure 42). In all the cases, we observe that there is a strong correlation between the degree of a vertex and its betweenness centrality score. All highly connected vertices have high centrality scores. However, observe that low-degree vertices show a significant variation in their centrality scores. The protein with the highest degree (P53) also has the highest centrality score.

We now try to explain the variation in the centrality scores of low-degree vertices. Clearly, all degree-1 vertices have a centrality score of zero. We would like to determine the connectivity patterns of high centrality, low-degree vertices in the graph. For this purpose, consider decomposing the graph into its biconnected components. These are the maximal subsets of vertices such that the removal of a vertex from a particular component will not disconnect the component. Unlike connected components, vertices may belong to multiple



(a) Human PIN



(b) Yeast PIN

Figure 40: Joint degree distributions of the human and yeast PINs.

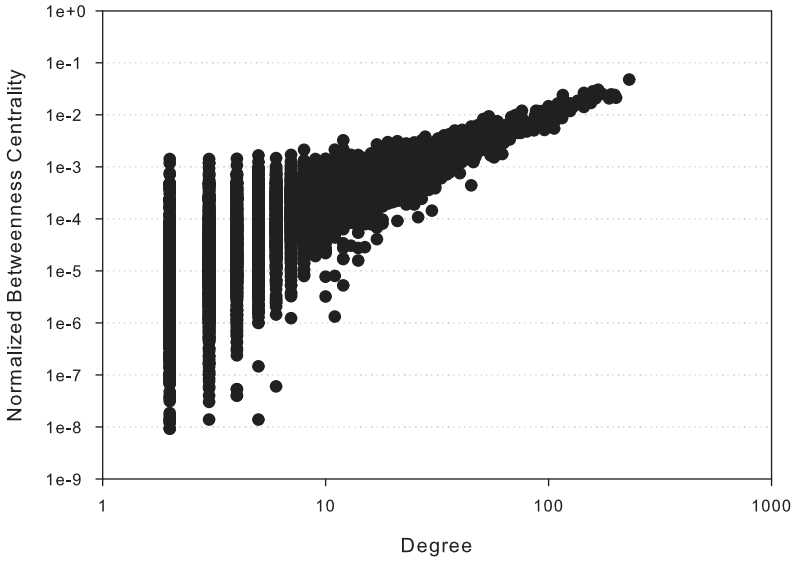
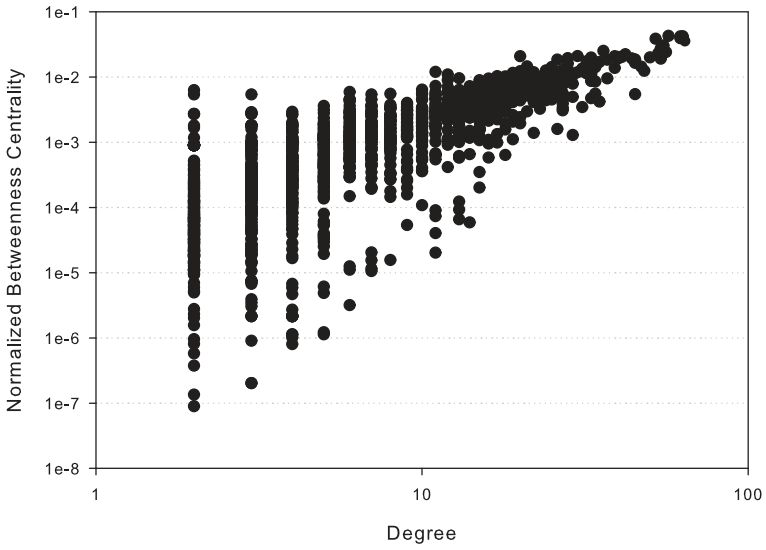


Figure 41: Normalized betweenness centrality vs. degree in HPIN.

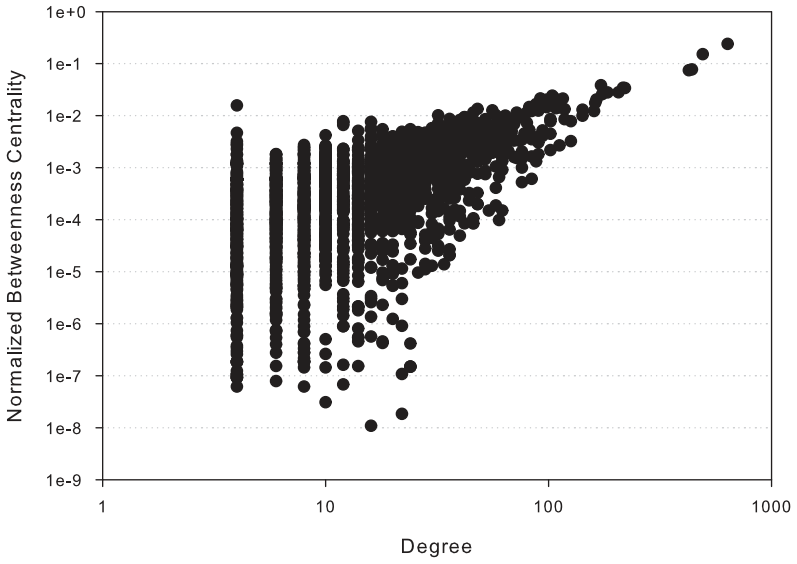
biconnected components: vertices that belong to more than one biconnected component are called *articulation points* or, equivalently, *cut vertices*. A graph without articulation points is biconnected. Figure 43 replots betweenness centrality scores of vertices in the graph, indicating articulation points separately this time. We observe that *nearly all articulation vertices have high centrality scores*. We can filter a significant percentage of vertices in the graph based on the observation that low degree vertices that are not articulation points have low centrality scores, and are unlikely to be critical.

Figure 44 plots the percentage of articulation points vs degree in the Human PIN. Observe that 20% of all low-degree vertices are articulation points. Similarly, a high percentage of high-degree vertices are again articulation points. We can filter non-articulation low-degree vertices and high-degree articulation vertices, as the centrality scores for these vertices are predictably low and high respectively.

We now plot betweenness centrality scores in the pruned graph (after removing non-articulation low-degree vertices). The average centrality scores for a given vertex degree are indicated on a linear scale in Figure 45, along with the maximum and minimum values. Observe that centrality scores only vary by two orders of magnitude in this case. For a given degree, the centrality scores vary by an order of magnitude in both HPIN and YPIN.

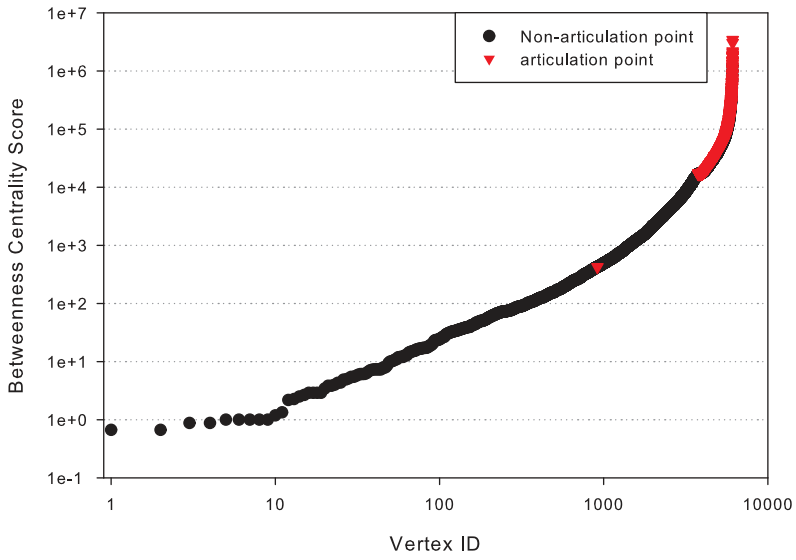


(a) Yeast PIN 1

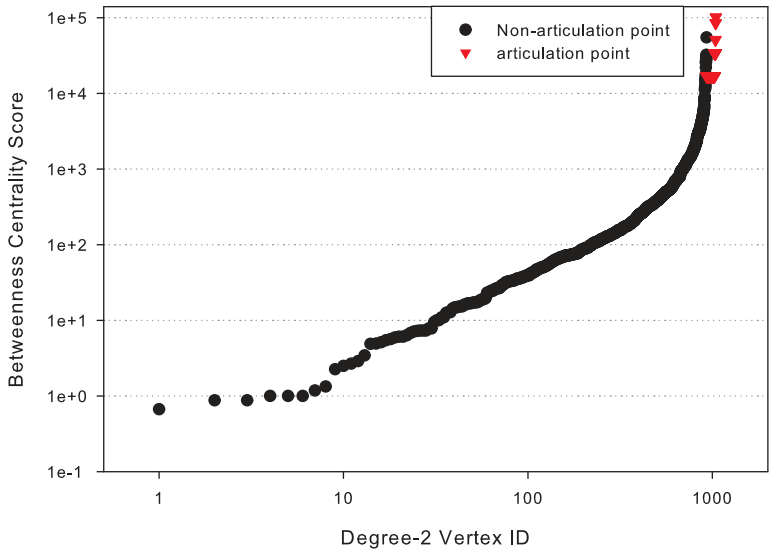


(b) Yeast PIN 2

Figure 42: Normalized betweenness centrality vs. degree in the yeast networks.



(a) Entire graph



(b) Only degree-2 vertices

Figure 43: Betweenness centrality scores of articulation and non-articulation vertices.

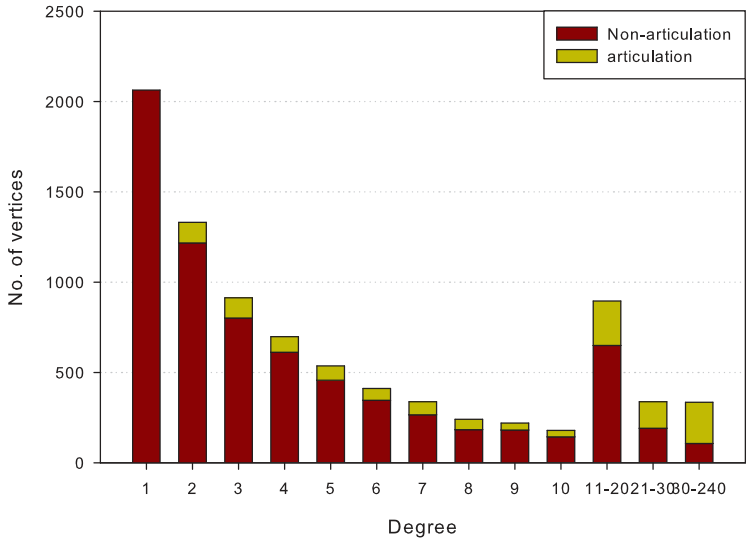


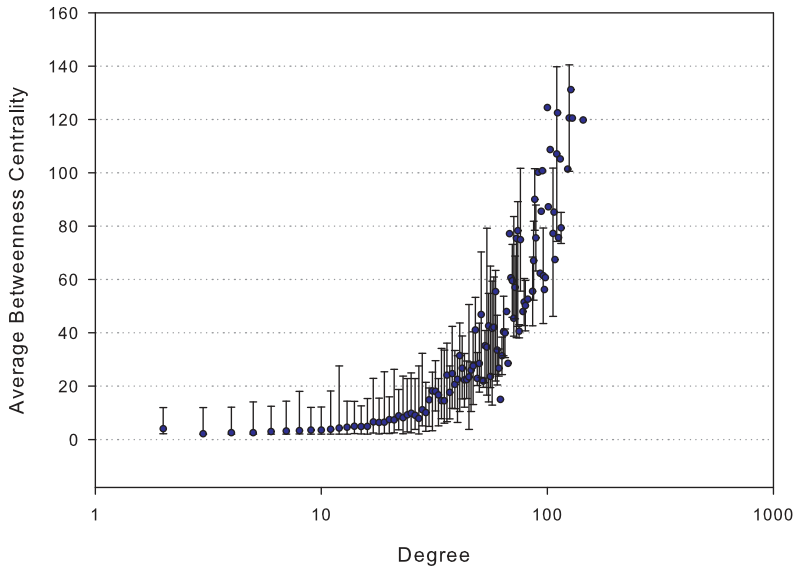
Figure 44: Percentage of articulation points in HPIN vs protein degree.

The centrality score variation is slightly higher in YPIN.

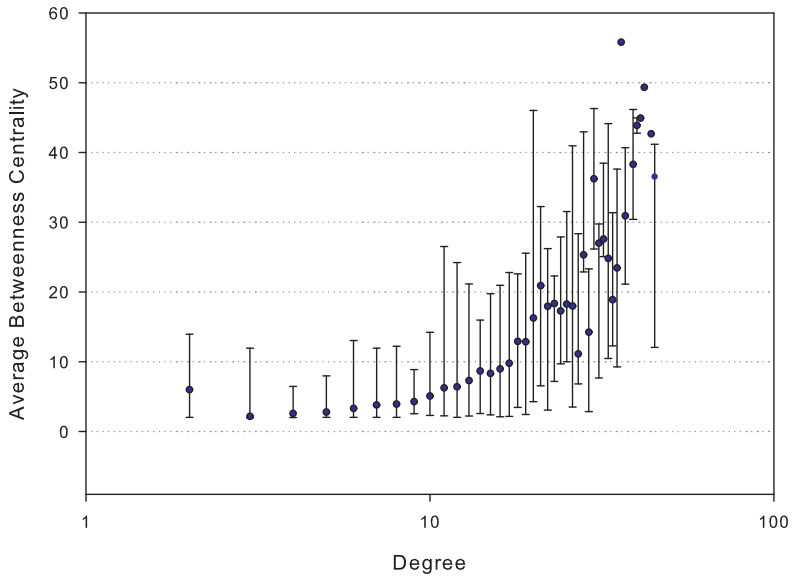
Based on analysis of the betweenness measure, we identify a new topological feature in the yeast and Human PINs that may not found in randomly generated scale-free networks: a significant percentage of proteins characterized by high betweenness (one order of magnitude less than the maximum), yet low connectivity. The fact that a majority of these proteins are also articulation points indicates suggests that these proteins may represent important that link components with a low degree of clustering.

We now investigate whether synthetic models for network evolution reproduce this low-degree, high betweenness behavior. To address this question, we analyzed different computational models of biological network evolution that generate scale-free networks. We experimented with a range of parameters for these models and selected ones that gave a power-law distribution that matched the slope of HPIN. In each case, we quantified the variation of betweenness for a particular connectivity, and its change for the value of connectivity. Thus, an increase in the standard deviation of betweenness values for low degree values indicated the presence of high centrality, low-degree proteins.

The simplest generative algorithm, first proposed by Barabasi and Albert (BA preferential attachment model) to explain the power-law distribution of connectivity, does not



(a) Human PIN



(b) Yeast PIN

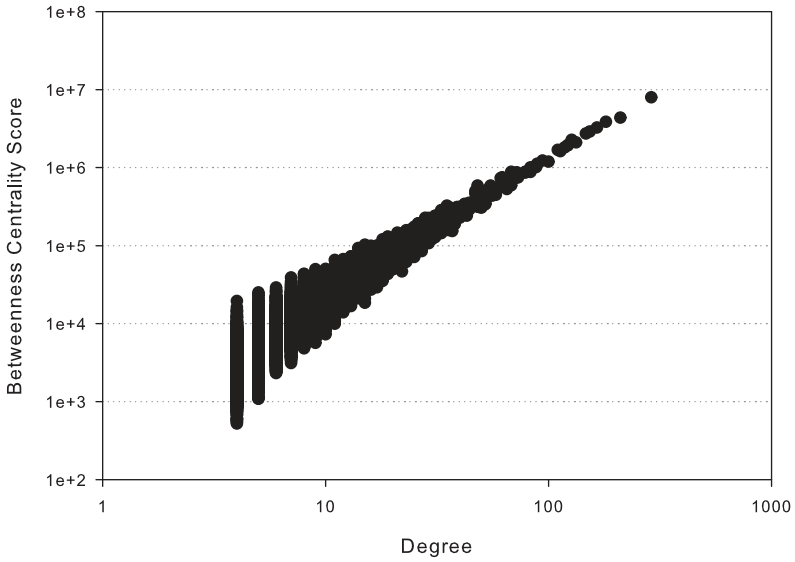
Figure 45: Average betweenness centrality vs. degree after removing low-degree non-articulation vertices (the error bars indicate the maximum and minimum values).

predict the existence of low-degree, high centrality vertices. Betweenness and degree are almost linearly correlated in this graph (see Figure 46(a)). Also, the extended Barabasi-Albert (EBA) model, where link addition and rewiring occur along with node addition with preferential attachment, also did not produce networks with this characteristic, although low-degree vertices showed a better spread in this case. Moreover, this algorithm has no biological basis.

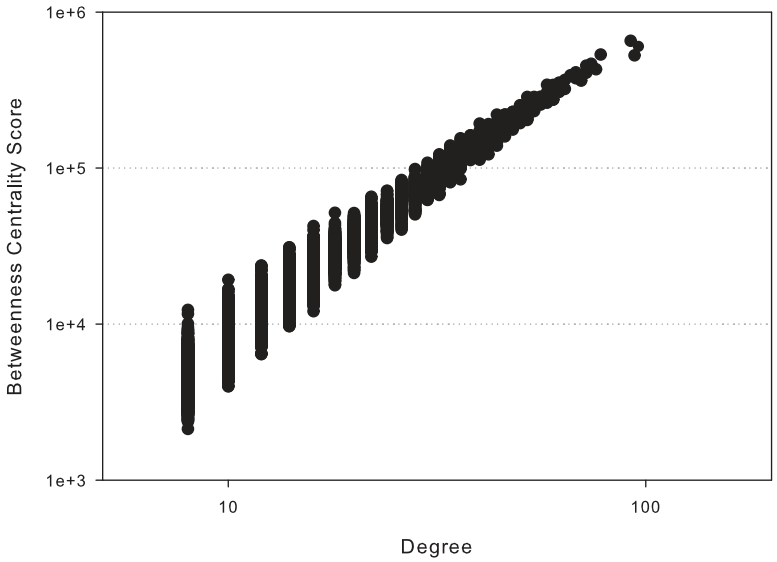
A biologically motivated model put forward by Sole et al. [171] and Vazquez et al. [180] incorporated *gene duplication* as the driving mechanism for genome growth. In this model, the existing nodes (proteins) are copied with all their existing links, followed by divergence of the duplicated nodes introduced by rewiring and/or addition of connections, imitating mutations of duplicated genes. For the model parameter range that produces power-law networks, the Sole-Vazquez (SV) model also failed to produce the same bias towards betweenness exhibited by HPIN (see Figure 46(b)). These results show that existing evolutionary algorithms that produce scale-free networks do not predict the existence of high-betweenness, low-degree vertices found within the yeast and human PINs.

Joy et al. [110] propose a model to explain a similar occurrence in the yeast PIN. They present a new model based on the Berg model [28], which considers point mutations in addition to gene duplication. With this model, the authors generate a synthetic network that matches the low degree, high centrality property in the yeast PIN. However, the paper [110] does not discuss algorithms for determining model parameters (the duplication and point mutation rates) to fit data given an arbitrary dataset, such as HPIN in our study. Thus, we were unable to determine whether this model could explain the centrality variance in HPIN.

Finally, we study the *coreness* of the graph using a simple heuristic. The k-core of a graph is defined as the subgraph obtained by recursively removing all vertices of degree less than k from the original graph. If a vertex belongs to the k-core but not to the $(k+1)$-core, we say that its vertex coreness is k. Vertices with degree-1 have $core = 0$. Coreness gives us an idea of how *deep* in the core a vertex is. It is related to vertex degree, but is a more sophisticated measure. A node with small coreness is not well connected and can be

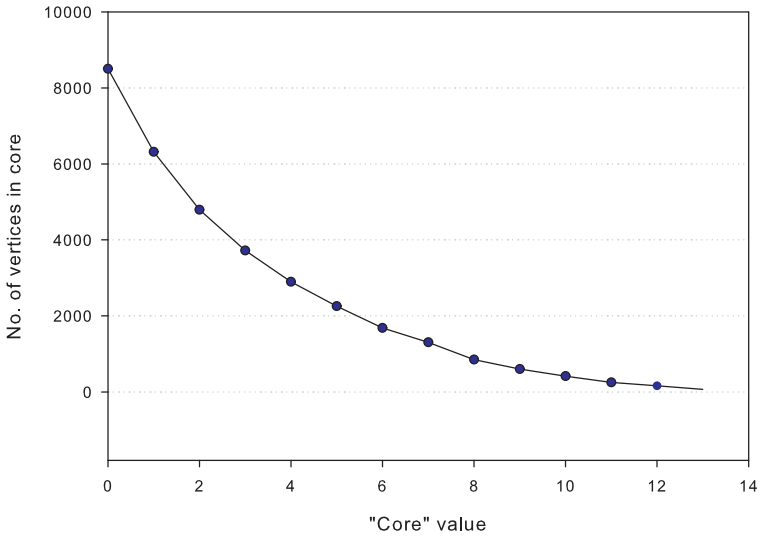


(a) Barabasi-Albert Preferential Attachment model

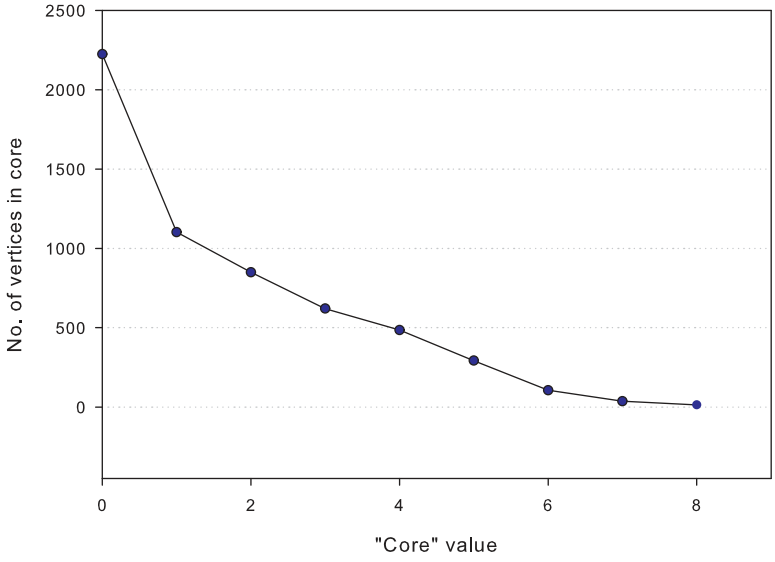


(b) Sole-Vazquez model

Figure 46: Degree-betweenness correlation for synthetic graphs that match the degree distribution of HPIN.



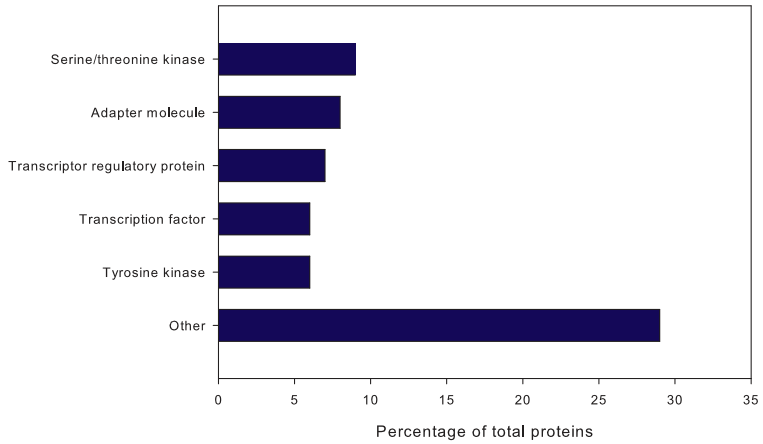
(a) Human PIN



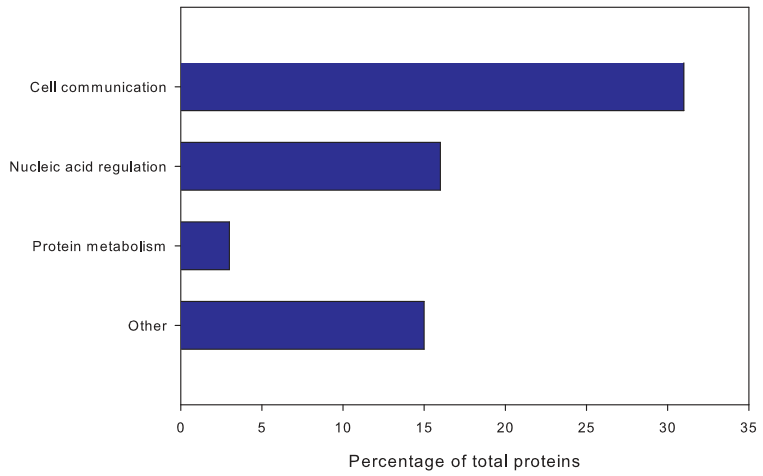
(b) Yeast PIN

Figure 47: Core-k distribution of the human and yeast PINs.

disconnected easily by removing its poorly connected neighbors, even if its degree is high. We see that this is exactly the case in both the human and yeast PINs (see Figure 47). This indicates an *absence of a high-degree core* in these PINs, which implies that the network may be vulnerable to attacks on select hub proteins.



(a) Molecular Class



(b) Biological Function

Figure 48: The dominant molecular classes and biological functions among proteins that are common to both the top 1% betweenness centrality and degree lists.

Figure 48 is a graphical representation of the dominant molecular class and biological function among high betweenness and high connectivity proteins (the common proteins in the top 1% lists). These proteins belong to a variety of molecular classes (Figure 48(a)),

with cell communication and signal transduction being the most common biological function (Figure 48(b)). The functional annotations are derived from Gene Ontology data. Due to the lack of comprehensive functional annotation data sets for the human interactome and software tools to process human interaction datasets, we could not complete functional enrichment tests on HPIN.

3.7 Betweenness conjecture for an Integer torus

Parallel implementations of betweenness centrality on large-scale irregular networks are often prone to programming error, and are computationally expensive to validate. Since the integer torus is a regular network that is easy to generate, we propose using it as a test instance for validating betweenness implementations, and in particular an analysis kernel of the HPCS SSCA graph theory benchmark [13]. We derive an analytical expression for exact betweenness for an integer torus in this section. An extended version of the proof appears in [10].

For $n \in \mathbb{N}$, let $\mathcal{T}_n$ denote an integer torus, that is the two-dimensional integer lattice mod n. Based on empirical evidence from extensive computational experimentation, we had the following:

Conjecture 1. *For $v \in \mathcal{T}_n$,*

$$BC(v) = \frac{n^3}{2} - n^2 - \frac{n}{2} + 1 \text{ when } n \text{ is odd,} \tag{1}$$

and

$$BC(v) = \frac{n^3}{2} - n^2 + 1 \text{ when } n \text{ is even.} \tag{2}$$

There is a parity dependence because of the impact of geodesics whose horizontal and/or vertical distance is maximal when n is even. For $s, t \in \mathcal{T}_n$ with $s = (x, y)$ and $t = (x', y')$, let

$$d_H(s,t) = \min\{(x' - x) \bmod n, (x - x') \bmod n\},$$

$$d_V(s,t) = \min\{(y' - y) \bmod n, (y - y') \bmod n\}, \text{ and}$$

$$d(s,t) = d_H(s,t) + d_V(s,t)$$

denote the horizontal, vertical, and total distance, respectively, from s to t. Note that $0 \leq d_H(s,t), d_V(s,t) \leq \frac{n}{2}$. When n is even and $d_H(s,t) = \frac{n}{2}$, then we say that s and t are horizontal **diameter achieving**. Similarly, when $d_V(s,t) = \frac{n}{2}$, then s and t are said to be vertical diameter achieving.

For $0 \leq d_H(s,t), d_V(s,t) < \frac{n}{2}$, we have that

$$\sigma_{st} = \binom{d_H(s,t) + d_V(s,t)}{d_V(s,t)} = \binom{d_H(s,t) + d_V(s,t)}{d_H(s,t)}.$$

If n is even and s and t are either horizontal or (exclusively) vertical diameter achieving, then

$$\sigma_{st} = 2\binom{d_H(s,t) + d_V(s,t)}{d_V(s,t)}.$$

If n is even and s and t are both horizontal and vertical diameter achieving, then

$$\sigma_{st} = 4\binom{d_H(s,t) + d_V(s,t)}{d_V(s,t)}.$$

Let $v = (p,q)$ for $p, q \in \mathbb{N}$ with $0 \leq p, q < n$. Observe that there is a shortest path between s and t which passes through v if and only if

$$d_H(s,t) = d_H(s,v) + d_H(v,t) \text{ and } d_V(s,t) = d_V(s,v) + d_V(v,t).$$

Since $\sigma_{st}(v) = \sigma_{sv}\sigma_{vt}$, we will calculate $BC(v)$ by counting all geodesics from s to v and from v to t where $\sigma_{st}(v)$ is nonzero. We exploit the symmetries of $\mathcal{T}_n$, and enumerate shortest paths through $v_0 = (0,0)$ for particular subsets of $s, t \in \mathcal{T}_n$. For $S, T \subseteq \mathcal{T}_n$ and $v_0 = (0,0)$, let

$$\Delta(S,T) = \sum_{s \in S, t \in T, s \neq v \neq t} \delta_{st}(v_0).$$

Let $m = \lfloor \frac{n}{2} \rfloor$ and $x, y \in \{-m, \ldots, 0, \ldots, m\}$ for $(x,y) \in \mathcal{T}_n$. We divide $\mathcal{T}_n$ into four quadrants, centered at v_0:

$$\begin{aligned}
Q_1 &= \{(-x,-y) \in \mathcal{T}_n \mid 0 \leq x, y \leq m\} \\
Q_2 &= \{(-x,y) \in \mathcal{T}_n \mid 0 \leq x, y \leq m\} \\
Q_3 &= \{(x,y) \in \mathcal{T}_n \mid 0 \leq x, y \leq m\} \\
Q_4 &= \{(x,-y) \in \mathcal{T}_n \mid 0 \leq x, y \leq m\}.
\end{aligned}$$

Since exactly which subsets of $\mathcal{T}_n$ are used depends on the parity of n, we consider the two cases separately.

3.7.1 Proof of conjecture when n is odd

We first prove Equation 1 of the conjecture, since there are no diameter achieving pairs of vertices when n is odd.

Lemma 7. *For $v_0 = (0,0) \in \mathcal{T}_n$ with n odd,*

$$\begin{aligned} BC(v_0) &= \Delta(Q_1,Q_3) + \Delta(Q_3,Q_1) + \Delta(Q_2,Q_4) + \Delta(Q_4,Q_2) \\ &\quad -\Delta(Q_1,Q_2) - \Delta(Q_3,Q_4) - \Delta(Q_1,Q_4) - \Delta(Q_2,Q_3) \\ &= 4 \cdot \Delta(Q_1,Q_3) - 4 \cdot \Delta(Q_1,Q_2). \end{aligned}$$

Proof. In the calculation of $BC(v_0)$, we sum over all possible pairs s and t, where a path from s and to t is counted as distinct from the reversed path which begins at t and ends at s. Thus, the first four terms count all possible paths from s to t along the four "diagonals" through v_0, with redundancy. By the symmetries of $\mathcal{T}_n$, these four values are all equal to $\Delta(Q_1,Q_3)$ which counts the fraction of shortest paths through v_0 from an $s \in Q_1$, the "lower-left" quadrant (modulo n) with respect to $(0,0)$, to t in the upper-right quadrant Q_3. The remaining additive factors then correct for the overcounting of geodesics along the vertical and horizontal, respectively, lines through v_0. Like before, these are all equal to $\Delta(Q_1,Q_2)$ where $\sigma_{st}(v_0) \neq 0$ if and only if $x = 0 = x'$ for $(x,y) \in Q_1$ and $(x',y') \in Q_2$.

Note that each of the paths where one of s and t lies on the horizontal line through v_0 and the other lies on the vertical line through v_0 is counted exactly twice (once in each direction) in $4 \cdot \Delta(Q_1,Q_2)$. For instance, $\Delta(Q_1,Q_2)$ counts paths where s lies on the horizontal line to the left of v_0 (again modulo n) or on the vertical line below v_0 and where t lies on the horizontal line to the right of v_0 or on the vertical line above v_0.

Theorem 4. *Let n be an odd integer. Suppose $v_0 = (0,0) \in \mathcal{T}_n$ and $m = \lfloor \frac{n}{2} \rfloor$, with*

$$\begin{aligned} Q_1 &= \{(-x,-y) \in \mathcal{T}_n \mid 0 \le x, y \le m\}, \\ Q_2 &= \{(-x,y) \in \mathcal{T}_n \mid 0 \le x, y \le m\}, \\ Q_3 &= \{(x,y) \in \mathcal{T}_n \mid 0 \le x, y \le m\}. \end{aligned}$$

Then

$$\Delta(Q_1, Q_2) = \frac{m(m-1)}{2} \tag{3}$$

and

$$\Delta(Q_1, Q_3) = 1 + (m-1)(m+1)^2. \tag{4}$$

Proof. Let $v_0 = (0,0) \in \mathcal{T}_n$ for an odd integer n. Consider $s, t \in \mathcal{T}_n$ where $s \neq v_0 \neq t$ and v_0 lies on a shortest path from s to t. Let $m = \lfloor \frac{n}{2} \rfloor$ and

$$\begin{aligned} d_H(s,t) &= h & d_V(s,t) &= k \\ d_H(s,v_0) &= i & d_V(s,v_0) &= j \\ d_H(v_0,t) &= i' = h - i & d_V(v_0,t) &= j' = k - j \end{aligned}$$

for $0 \le i \le h \le m < \frac{n}{2}$ and $0 \le j \le k \le m < \frac{n}{2}$ where not both $i = j = 0$ nor $i = h, j = k$ nor $h = k = 0$. In this notation, we have that

$$\sigma_{sv_0} = \binom{i+j}{i}, \ \sigma_{v_0 t} = \binom{(h-i)+(k-j)}{(h-i)}, \text{ and } \sigma_{st} = \binom{h+k}{h}.$$

Suppose further that $s \in Q_1$ and $t =\in Q_2$. Since $\sigma_{st}(v_0) = 0$ unless $x = 0 = x'$, we need only consider $i = 0$. Since $j = 0$ implies that $s = v_0$ and $j = k$ implies that $t = v_0$, we consider only $0 < j < k$. When $i = 0$ and $0 < j < k \le m < \frac{n}{2}$, then $\sigma_{st} = 1$ and $\sigma_{st}(v_0) = 1$. Thus, since $i = 0$ and $h = 0$,

$$\begin{aligned} \Delta(Q_1, Q_2) &= \sum_{k=2}^{m} \sum_{j=1}^{k-1} \frac{\binom{0+j}{0}\binom{0+(k-j)}{0}}{\binom{0+k}{0}} \\ &= \frac{m(m-1)}{2} \end{aligned}$$

where we begin the outer summation at $k = 2$ since $h = 0$, $k = 0$ implies that $s = v_0 = t$ and $h = 0, k = 1$ implies that either $s = v_0$ or $t = v_0$. Hence, Equation 3 in Theorem 4 holds.

Suppose now that $s \in Q_1$ and $t \in Q_3$. As with our previous equality, we enumerate $\Delta(Q_1, Q_3)$ by summing over the possible values of $0 \leq i \leq h \leq m < \frac{n}{2}$ and $0 \leq j \leq k \leq m < \frac{n}{2}$ where not both $i = j = 0$ nor $i = h, j = k$ nor $h = k = 0$. Also, as before, if $h = 0$, then $k > 1$ and vice versa. Thus,

$$\Delta(Q_1, Q_3) = 1 + \sum_{h=0}^{m}\sum_{k=0}^{m}(-2 + \frac{1}{\binom{h+k}{k}}\sum_{i=0}^{h}\sum_{j=0}^{k}\binom{i+j}{i}\binom{(h-i)+(k-j)}{(h-i)})$$

where we have corrected for $i = 0 = j$ and $h - i = 0 = k - j$ by subtracting out the terms $\binom{0+0}{0}\binom{h+k}{k}$ and $\binom{h+k}{k}\binom{0+0}{0}$. Likewise, we have corrected for $h = 0, k = 0$ by adding 1. Note that when $h = 0, k = 1$ and $h = 1, k = 0$, the expression inside the h and k summands is zero and no correction is needed. We know that

$$\sum_{j=0}^{k}\binom{i+j}{i}\binom{(h-i)+(k-j)}{(h-i)}) = \binom{h+k+1}{k},$$

either as an application of Equation 5.26 from [86] (found via identity # 3100005 on the Pascal's Triangle website `http://binomial.csuhayward.edu/`) or proved directly via induction and parallel summation. Hence, we have that

$$\begin{aligned}
\Delta(Q_1, Q_3) &= 1 + \sum_{h=0}^{m}\sum_{k=0}^{m}(-2 + \frac{1}{\binom{h+k}{k}}\sum_{i=0}^{h}\binom{h+k+1}{k}) \\
&= 1 + \sum_{h=0}^{m}\sum_{k=0}^{m}(-2 + \frac{1}{\binom{h+k}{k}}(h+1)\frac{h+k+1}{h+1}\binom{h+k}{k}) \\
&= 1 + \sum_{h=0}^{m}\sum_{k=0}^{m}(h+k-1) \\
&= 1 + (m-1)(m+1)^2
\end{aligned}$$

and Equation 4 of Theorem 4 also holds.

Since $m = \frac{n-1}{2}$ and

$$BC(v_0) = 4 \cdot \Delta(Q_1, Q_3) - 4 \cdot \Delta(Q_1, Q_2) = BC(v) \text{ for all } v \in \mathcal{T}_n,$$

as an immediate consequence of Theorem 4 we have that

Corollary 1. *When n is odd,*

$$BC(v) = \frac{n^3}{2} - n^2 - \frac{n}{2} + 1.$$

3.7.2 Proof of conjecture when n is even

The proof of Equation 2 of the conjecture is similar, although considerably more complicated when n is even and $m = \frac{n}{2}$. The complications are due to any diameter achieving pairs $s, t \in \mathcal{T}_n$, which double the number of geodesics when either $d_H(s,t)$ or $d_V(s,t) = \frac{n}{2}$ and quadruple it when s and t are both horizontal and vertical diameter achieving.

Again, we let

$$\begin{aligned} Q_1 &= \{(-x,-y) \in \mathcal{T}_n \mid 0 \le x, y \le m\} \\ Q_2 &= \{(-x,y) \in \mathcal{T}_n \mid 0 \le x, y \le m\} \\ Q_3 &= \{(x,y) \in \mathcal{T}_n \mid 0 \le x, y \le m\} \\ Q_4 &= \{(x,-y) \in \mathcal{T}_n \mid 0 \le x, y \le m\}. \end{aligned}$$

Also, for $s, t, v_0 = (0,0) \in \mathcal{T}_n$, we will use the same notation of

$$\begin{aligned} &d_H(s,t) = h && d_V(s,t) = k \\ &d_H(s,v_0) = i && d_V(s,v_0) = j \\ &d_H(v_0,t) = i' = h - i && d_V(v_0,t) = j' = k - j \end{aligned}$$

for $0 \le i \le h \le m = \frac{n}{2}$ and $0 \le j \le k \le m = \frac{n}{2}$ where not both $i = j = 0$ nor $i = h, j = k$ nor $h = k = 0$.

When n was odd, we were able to compute $BC(v_0)$ as a function only of $\Delta(Q_1, Q_3)$ and $\Delta(Q_1, Q_2)$. Now that n is even, the basic approach is the same except that we must consider a number of different subcases due to the impact of the diameter achieving pairs on the enumeration of

$$\delta_{st}(v_0) = \frac{\sigma_{st}(v_0)}{\sigma_{st}} = \frac{\sigma_{sv_0}\sigma_{v_0t}}{\sigma_{st}}.$$

In our notation, we still have that

$$\sigma_{sv_0} = \binom{i+j}{i},\ \sigma_{v_0t} = \binom{(h-i)+(k-j)}{(h-i)},\ \text{and } \sigma_{st} = \binom{h+k}{h}$$

when $0 \le i \le h < m = \frac{n}{2}$, $0 \le j \le k < m = \frac{n}{2}$. If instead $h = m$ but $i, h-i \ne m$, then we have

$$\sigma_{sv_0} = \binom{i+j}{i},\ \sigma_{v_0t} = \binom{(h-i)+(k-j)}{(h-i)},\ \text{and } \sigma_{st} = 2\binom{h+k}{h}.$$

By the symmetry between i and j, h and k, this is also true for $k = m$ and $j, k - j \neq m$. When $h = m$ and $i = 0$, $h = m$ and $i = m$, $k = m$ and $j = 0$, or $k = m$ and $j = m$, then

$$\delta_{st}(v_0) = \frac{2\binom{i+j}{i}\binom{(h-i)+(k-j)}{(h-i)}}{2\binom{h+k}{h}}.$$

When $h = k = m$, then s and t are both horizontal and vertical diameter achieving. If $1 \leq i \leq m-1$ and $1 \leq j \leq m-1$, then

$$\sigma_{sv_0} = \binom{i+j}{i},\ \sigma_{v_0t} = \binom{(h-i)+(k-j)}{(h-i)},\ \text{and}\ \sigma_{st} = 4\binom{h+k}{h}.$$

If exactly one of i, $h-i$, j, or $k-j$ is also diameter achieving, then

$$\delta_{st}(v_0) = \frac{2\binom{i+j}{i}\binom{(h-i)+(k-j)}{(h-i)}}{4\binom{h+k}{h}}$$

while if both i and j or both $h-i$ and $k-j$ are diameter achieving then

$$\delta_{st}(v_0) = \frac{4\binom{i+j}{i}\binom{(h-i)+(k-j)}{(h-i)}}{4\binom{h+k}{h}}.$$

If we enumerate $\delta_{st}(v_0)$ for all these different cases, then we will be able to calculate $BC(v_0)$ as we did before.

Theorem 5. *Let n be an even integer. Suppose $v_0 = (0,0) \in \mathcal{T}_n$ and $m = \frac{n}{2}$. Then*

$$\begin{aligned} BC(v_0) &= 4 \cdot \Delta(S_1, T_1) && (5)\\ &+ 4 \cdot 2 \cdot \Delta(S_2, T_2) && (6)\\ &+ 2 \cdot 4 \cdot \Delta(S_3, T_3) && (7)\\ &+ 4 \cdot \Delta(S_4, T_4) && (8)\\ &+ 2 \cdot 4 \cdot \Delta(S_5, T_5) && (9)\\ &+ 2 \cdot \Delta(S_6, T_6) && (10)\\ &- 4 \cdot \Delta(S_7, T_7) && (11) \end{aligned}$$

where, for $d_H(s, v_0) = i$, $d_V(s, v_0) = j$, $d_H(s, t) = h$, *and* $d_V(s, t) = k$,

$$
\begin{aligned}
S_1 &= \{s \in Q_1 \mid 0 \le i \le h \le m-1, 0 \le j \le k \le m-1\} \\
T_1 &= \{t \in Q_3 \mid 0 \le h-i \le h \le m-1, 0 \le k-j \le k \le m-1\},
\end{aligned}
$$

$$
\begin{aligned}
S_2 &= \{s \in Q_1 \mid 1 \le i < h = m, 0 \le j \le k \le m-1\} \\
T_2 &= \{t \in Q_3 \mid 1 \le h-i < h = m, 0 \le k-j \le k \le m-1\},
\end{aligned}
$$

$$
\begin{aligned}
S_3 &= \{s \in Q_1 \mid i = 0, 0 \le j \le k \le m-1\} \\
T_3 &= \{t \in Q_3 \mid h-i = m, 0 \le k-j \le k \le m-1\},
\end{aligned}
$$

$$
\begin{aligned}
S_4 &= \{s \in Q_1 \mid 1 \le i < h = m, 1 \le j < k = m\} \\
T_4 &= \{t \in Q_3 \mid 1 \le h-i < h = m, 1 \le k-j < k = m\},
\end{aligned}
$$

$$
\begin{aligned}
S_5 &= \{s \in Q_1 \mid i = 0, 1 \le j < k = m\} \\
T_5 &= \{t \in Q_3 \mid h-i = m, 1 \le k-j < k = m\},
\end{aligned}
$$

$$
\begin{aligned}
S_6 &= \{s \in Q_1 \mid i = 0, j = m\} \\
T_6 &= \{t \in Q_3 \mid h-i = m, k-j = 0\},
\end{aligned}
$$

$$
\begin{aligned}
S_7 &= \{s \in Q_1 \mid i = 0, 1 \le j < k = m\} \\
T_7 &= \{t \in Q_3 \mid h-i = 0, 1 \le k-j < k = m\}
\end{aligned}
$$

and in each case not both $i = j = 0$ *nor* $i = h, j = k$ *nor* $h = k = 0$.

Proof. The different pairs S_i, T_i for $1 \leq i \leq 7$ correspond to the following cases:

S_1, T_1	$0 \leq h \leq m-1$	$0 \leq k \leq m-1$	$0 \leq i \leq h$	$0 \leq j \leq k$
S_2, T_2	$h = m$	$0 \leq k \leq m-1$	$1 \leq i \leq h-1$	$0 \leq j \leq k$
	$0 \leq h \leq m-1$	$k = m$	$0 \leq i \leq h$	$1 \leq j \leq k-1$
S_3, T_3	$h = m$	$0 \leq k \leq m-1$	$i = 0$	$0 \leq j \leq k$
	$h = m$	$0 \leq k \leq m-1$	$i = m$	$0 \leq j \leq k$
	$0 \leq h \leq m-1$	$k = m$	$0 \leq i \leq h$	$j = 0$
	$0 \leq h \leq m-1$	$k = m$	$0 \leq i \leq h$	$j = m$
S_4, T_4	$h = m$	$k = m$	$1 \leq i \leq h-1$	$1 \leq j \leq k-1$
S_5, T_5	$h = m$	$k = m$	$i = 0$	$1 \leq j \leq k-1$
	$h = m$	$k = m$	$i = m$	$1 \leq j \leq k-1$
	$h = m$	$k = m$	$1 \leq i \leq h-1$	$j = 0$
	$h = m$	$k = m$	$1 \leq i \leq h-1$	$j = m$
S_6, T_6	$h = m$	$k = m$	$i = 0$	$j = m$
S_7, T_7	$h = 0$	$2 \leq k \leq m$	$i = 0$	$1 \leq j < k = m$

For $S_1 \subset Q_1, T_1 \subset Q_3$ and $S_2 \subset Q_1, T_2 \subset Q_3$, there are corresponding distinct sets in each of Q_2, Q_4, and Q_3, Q_1, and Q_4, Q_2. This is also true for S_4, T_4. For S_3 and T_3, however, we also have that $S_3 \subset Q_4$ and $T_3 \subset Q_2$. Thus, we only multiply $\Delta(S_3, T_3)$ by a factor of two to account for the reverse paths from $Q_3 \cap Q_2$ to $Q_1 \cap Q_4$. This is also the case for S_5 and T_5. For S_6, T_6, we first note that $i = m, j = 0$ gives the same path, just in the opposite direction. Since this is the only such path, it is counted twice. Finally, S_7, T_7 correct for the overcounting along the horizontal and vertical lines through v_0, once in each direction, for a total factor of four.

Theorem 6. *Suppose the assumptions of Theorem 5 hold. Then*

$$\Delta(S_1, T_1) = 1 + (m-2)m^2 \tag{12}$$

$$\Delta(S_2, T_2) = \frac{(m-1)m(3m+1)}{4(m+1)} \tag{13}$$

$$\Delta(S_3, T_3) = \frac{(m-1)m}{2(m+1)} \tag{14}$$

$$\Delta(S_4, T_4) = \frac{1}{4\binom{m+m}{m}}((m-3)\binom{2m+1}{m} + 2(1) + 2\binom{2m}{m}) \tag{15}$$

$$\Delta(S_5, T_5) = \frac{\binom{m+m+1}{m} - 1 - \binom{2m}{m}}{2\binom{2m}{m}} \tag{16}$$

$$\Delta(S_6, T_6) = \frac{1}{\binom{2m}{m}} \tag{17}$$

$$\Delta(S_7, T_7) = \frac{(m-1)^2}{2} \tag{18}$$

Proof. The result follows from the following summations, and extensive use of the equality on page 126.

$$\begin{aligned}
\Delta(S_1, T_1) &= 1 + \sum_{h=0}^{m-1}\sum_{k=0}^{m-1}(-2 + \frac{1}{\binom{h+k}{k}}\sum_{i=0}^{h}\sum_{j=0}^{k}\binom{i+j}{i}\binom{(h-i)+(k-j)}{(h-i)}) \\
\Delta(S_2, T_2) &= \sum_{k=0}^{m-1}\frac{1}{2\binom{m+k}{k}}\sum_{i=1}^{m-1}\sum_{j=0}^{k}\binom{i+j}{i}\binom{(m-i)+(k-j)}{(m-i)} \\
\Delta(S_3, T_3) &= \sum_{k=0}^{m-1}(-1 + \frac{1}{\binom{m+k}{k}}\sum_{j=0}^{k}\binom{0+j}{0}\binom{(m-0)+(k-j)}{(m-0)}) \\
\Delta(S_4, T_4) &= \frac{1}{4\binom{m+m}{m}}\sum_{i=1}^{m-1}\sum_{j=1}^{m-1}\binom{i+j}{i}\binom{(m-i)+(m-j)}{(m-i)} \\
\Delta(S_5, T_5) &= \frac{1}{2\binom{m+m}{m}}\sum_{j=1}^{m-1}\binom{0+j}{0}\binom{(m-0)+(m-j)}{(m-0)} \\
\Delta(S_6, T_6) &= \frac{1}{\binom{m+m}{m}}\binom{0+m}{0}\binom{(m-0)+(m-m)}{(m-0)} \\
\Delta(S_7, T_7) &= \sum_{k=2}^{m-1}\sum_{j=1}^{k-1}\frac{\binom{0+j}{0}\binom{0+(k-j)}{0}}{\binom{0+k}{0}} + \sum_{j=1}^{m-1}\frac{\binom{0+j}{0}\binom{0+(m-j)}{0}}{2\binom{0+m}{0}}
\end{aligned}$$

As an immediate consequence of Theorem 5 and Theorem 6, we have

Corollary 2. *When n is even,*

$$BC(v) = \frac{n^3}{2} - n^2 + 1.$$

3.8 Summary

In this chapter, we present fast parallel algorithms for centrality analysis in real-world networks. The algorithms exploit common topological properties of small-world networks such as the low diameter and the unbalanced degree distributions. We compute exact betweenness centrality for several large networks such as web crawls, protein-interaction networks, movie-actor and patent citation networks. These graph instances are *three orders of magnitude larger* than the problem sizes that can be processed by current social network analysis packages. We also achieve impressive parallel performance on several multicore, symmetric multiprocessor and multithreaded systems. The centrality implementations are part of the open-source framework SNAP.

We present a novel approximation algorithm for computing betweenness centrality of a given vertex, in both weighted and unweighted graphs. Our approximation algorithm is based on an adaptive sampling technique that significantly reduces the number of single-source shortest path computations for vertices with high centrality. We conduct an extensive experimental study on real-world graph instances, and observe that the approximation algorithm performs well on web crawls, road networks and biological networks. Approximating the centrality of *all* vertices in time less than $O(nm)$ for unweighted graphs and $O(nm + n^2 \log n)$ for weighted graphs is an open problem. Also, designing a fully dynamic algorithm for computing betweenness is a challenging research problem.

In Section 3.6, we conduct an extensive study of the global topological characteristics of the human protein interaction network. We report a new topology feature in the yeast and human PINs not found in synthetic scale-free networks: the prevalence of low degree proteins with high-betweenness values. The high-betweenness, low centrality vertices also provide some insight into the clustering nature and coreness of the network. We find that vertices with high centrality scores are very likely to be articulation points in the graph, and also have low clustering coefficients. The source code for the various graph analysis routines is freely available online from our web site. We also intend to provide our centrality analysis codes as plug-ins to the biological network visualization tool Cytoscape [167].

CHAPTER IV

THE SNAP FRAMEWORK AND COMMUNITY IDENTIFICATION

A key problem in social network analysis is that of finding communities, dense components, or detecting other latent structure. This is usually formulated as a graph clustering problem, and several indices have been proposed for measuring the quality of clustering (see [112, 33] for a review). Existing approaches based on the Kernighan-Lin algorithm [117], spectral algorithms [112], flow-based algorithms, and hierarchical clustering work well for specific classes of networks (e.g., abstractions from scientific computing, physical topologies), but perform poorly for small-world networks. Newman and Girvan recently proposed a divisive algorithm based on edge betweenness [147] that has been applied successfully to a variety of real networks. However, it is compute-intensive and takes $O(n^3)$ time for sparse graphs (n denotes the number of vertices). This algorithm optimizes for a novel clustering measure called modularity, which has become very popular for social network analysis. We design three clustering schemes (two hierarchical agglomerative approaches, and one divisive clustering algorithm) that exploit typical topological characteristics of small-world networks. We also conduct an extensive experimental study and demonstrate that our parallel schemes give significant running time improvements over existing modularity-based clustering heuristics. For instance, our novel divisive clustering approach based on approximate edge betweenness centrality is *more than two orders of magnitude* faster than the Newman-Girvan algorithm on the Sun Fire T2000 multicore system, while maintaining comparable clustering quality.

4.1 Graph Partitioning

Graph partitioning and community detection are related problems, but with an important difference: the most commonly used objective function in partitioning is minimization of edge cut, while trying to *balance the number of vertices* in each partition. The number of partitions is typically an input parameter for a partitioning algorithm. Clustering, on

the other hand, optimizes an appropriate application-dependent measure, and the number of clusters needs to be computed. Multi-level algorithms and spectral heuristics have been shown to be very effective for partitioning graph abstractions derived from physical topologies, such as finite-element meshes arising in scientific computing. Software packages implementing these algorithms (e.g., Chaco [95] and Metis [115, 114]) are freely available, computationally efficient, and produce high-quality partitions in most cases. A natural question that arises is whether these partitioning algorithms, or simple variants, can be applied to small-world networks as well.

Graph Instance	*Metis-kway*	*Metis-recur*	*Chaco-RQI*	*Chaco-LAN*
Physical (road)	1,856	1,703	2,937	3,913
Sparse random	685,211	706,625	717,960	737,747
Small-world	805,903	736,560	–	–

Table 5: Performance results (edge cut) for a 32-way partitioning of three different graph instances, using standard partitioning algorithms from the Chaco and Metis packages. Chaco-RQI and Chaco-LAN fail to complete for the small-world network instance.

Table 5 summarizes results from an experiment to test the quality of existing partitioning packages on small-world networks. We consider graph instances from three different families (a road network, a sparse random graph, and a synthetic small-world network), but of the same size: roughly 200,000 vertices and 1 million edges. We report the edge cut for a balanced 32-way partitioning of each of these graphs, using four partitioning techniques (the default multilevel partitioning approaches from Metis, pmetis and kmetis, and two spectral heuristics from Chaco). Observe that the edge cut for the random and power-law graphs is nearly two orders of magnitude higher than the cut for the nearly-Euclidean road network. Clearly, existing partitioning tools fail to partition small-world networks since these networks lack the topological regularity found in scientific meshes and physical networks, where the degree distribution is relatively constant and most connectivity is localized. Also, random and small-world networks have a lower diameter ($O(\log n)$, or in some cases $O(1)$) than physical networks (e.g., $O(\sqrt{n})$ for Euclidean topologies). Lang [124, 125] provides further empirical evidence that *cut quality varies inversely with cut balance* for social graphs such as the Yahoo! IM network and the DBLP collaboration data set. Further, he shows that the spectral method tends to break off small parts of the graphs. This finding

is corroborated by a recent theoretical result from Mihail and Papadimitriou [141]. They prove that for a random graph with a skewed degree distribution, the largest eigenvalues are in correspondence with high-degree vertices, and the corresponding eigenvectors are the characteristic vectors of their neighborhoods. Spectral analysis in this case ignores structural features of the graph in favor of high-degree vertices.

Recent research efforts have focused on adapting multilevel and spectral partitioning techniques to small-world graphs. Abou-Rjeili and Karypis [1] present new coarsening heuristics for multilevel approaches that outperform (give a lower edge cut) Metis and Chaco. As it is difficult to theoretically analyze general small-world networks, researchers have been looking at applying spectral analysis to synthetic graph models. For instance, Dasgupta et al. [52] provide a normalization of the Laplacian that improves the performance of the spectral approach on a planted-partition random graph model. Clustering heuristics based on the above graph partitioning algorithms optimize for conductance, a measure that compares the cut size to cut balance. However, based on empirical and theoretical evidence that it is difficult to obtain balanced partitions in small-world networks, we focus on optimizing *modularity* [147], a popular heuristic from the complex network analysis community.

4.1.1 Modularity as a clustering measure

Intuitively, modularity is a measure that is based on optimizing *intra-cluster density over inter-cluster sparsity* [33]. Let $C = (C_1, ..., C_k)$ denote a partition of V such that $C_i \neq \phi$ and $C_i \cap C_j = \phi$. We call C a clustering of G and each C_i is defined to be a *cluster*. The cluster $G(C_i)$ is identified with the induced subgraph $G[C_i] := (C_i, E(C_i))$, where $E(C_i) := \{\langle u, v\rangle \in E : u, v \in C_i\}$. Then, $E(C) := \cup_{i=1}^{k} E(C_i)$ is the set of intra-cluster edges and $\tilde{E}(C) := E - E(C)$ is the set of inter-cluster edges. Let $m(C_i)$ denote the number of inter-cluster edges in C_i. Then, the modularity measure $q(C)$ of a clustering C is defined as

$$q(C) = \sum_i \left[\frac{m(C_i)}{m} - \left(\frac{\sum_{v \in C_i} deg(v)}{2m} \right)^2 \right]$$

To maximize the first term, the number of intra-cluster edges should be high, whereas

the second term is minimized by splitting the graph into multiple clusters with small total degrees. If a particular clustering gives no more intra-community edges than would be expected by random chance, we will get $Q = 0$. Values greater than 0 indicate deviation from randomness, and empirical results show that values greater than 0.3 indicate significant community structure. Modularity has found widespread acceptance in the network analysis community, and there have been an array of heuristics, based on spectral analysis, simulated annealing, greedy agglomeration, and extremal optimization [146] proposed to optimize it. Brandes *et al.* [32] recently showed that maximizing modularity is strongly $\mathcal{NP}$-complete, and this has led to renewed interest in designing better algorithms for modularity maximization. We present three new modularity-maximization heuristics in Section 4.2 and compare them with the current state-of-the-art approaches discussed in [146].

4.2 Parallel Community Identification Algorithms

The parallel algorithms we present for community identification are based on modularity maximization. Intuitively, modularity captures the idea that a good division of a network into communities is one in which there are *fewer than expected* edges between communities, and not one that just minimizes edge cut. Since the general problem of modularity optimization is $\mathcal{NP}$-complete [33], we explore greedy strategies that maximize modularity. Existing algorithms fall into two broad classes, divisive or agglomerative, based on how the division is done. In the agglomerative method, each vertex initially belongs to a singleton community, and two communities whose amalgamation produces an increase in the modularity score are merged together. The agglomeration can be represented by a tree, referred to as a dendrogram, whose internal nodes correspond to joins. In the following discussion, we present three novel *parallel* algorithms, one divisive and two agglomerative approaches, that are built on top of optimized SNAP analysis kernels.

4.2.1 Approximate betweenness-based divisive algorithm (pBD)

Our first approach is a divisive algorithm in which we initially treat the entire network as one community, and iteratively determine *critical* links in the network that can be cut. By doing this repeatedly, we divide the network into smaller and smaller components, and

Algorithm 10: Approximate betweenness-based divisive algorithm (pBD).

Input: $G(V, E)$, length function $l : E \rightarrow \mathbb{R}$
Output: A partition $C = (C_1, ..., C_k)$ ($C_i \neq \phi$ and $C_i \cap C_j = \phi$) of V that maximizes modularity; A dendrogram D representing the clustering steps.

```
1 Optional step: Run biconnected components, identify articulation points and bridges.
2 numIter ⟵ 0;
3 while numIter < m do
4     Find edge e_m with the highest approximate betweenness centrality score in parallel.
5     Mark edge e_m as deleted in the graph G.
6     Run connected components on G, update dendrogram and number of clusters in parallel.
7     Compute modularity of the current partitioning in parallel.
8     numInter ⟵ numIter + 1;
9 Inspect the dendrogram, set C to the clustering with the highest modularity score.
```

can also keep track of the clustering quality at each step by computing the modularity score. Algorithm 10 gives the high-level pseudocode for this iterative approach, and our parallelization strategy. We explain each step in more detail below.

There are several possible approaches to select the *critical link* on each iteration. Newman and Girvan [146] suggest picking edges based on their betweenness score, and show that this approach results in significantly higher modularity scores compared to other known greedy heuristics. The problem with this approach is that it is computationally expensive — we need to recompute edge betweenness centrality scores at each step of the algorithm, and there can be $O(m)$ iterations in the worst case. Although it might be tempting to compute betweenness scores only once and then remove edges in that order, Newman and Girvan show that this results in inferior clustering quality for several small-world networks.

We rely on extensive algorithm engineering and parallelization to speed up the Newman-Girvan edge betweenness technique, while trying to maintain the quality of clustering. First, observe that on each iteration, we only need to identify the edge with the highest centrality score. We recently proposed a novel betweenness computation algorithm based on adaptive sampling [12] for estimating the centrality score of a *specific vertex or edge* in a general network. It is adaptive in that the number of samples (graph traversals) varies with the information obtained from each sample; further, we show high-probability bounds on the

estimated error. In practice, after extensive experimentation on real-world networks, we show that on an average, we can estimate betweenness scores of high-centrality (the top 1%) entities with less than 20% error, by sampling just 5% of the vertices. We replace the exact centrality computation algorithm with the approximate betweenness approach, and only recompute aproximate betweenness scores of the known high-centrality edges (step 4 of Algorithm 10).

A second effective optimization is to vary the granularity of parallelization as the clustering algorithm proceeds. In the initial iterations of the algorithm, before the graph is split up into connected components of smaller sizes, we parallelize computation of approximate betweenness centrality. Once the graph is decomposed into a large number of isolated components, we can switch to computing exact centrality. We can then exploit parallelism at a coarser granularity, by computing centrality scores of each component in parallel. This switch in the parallelism granularity is semi-automatic (controller by a user parameter) in our SNAP implementation. In addition, we parallelize the $O(m)$-work kernels such as modularity computation (step 7 of Algorithm 10) and dendrogram updates (step 6 of Algorithm 10). Note that varying the parallelization granularity does not affect the quality of clustering (the modularity score) in any manner.

From empirical evidence, we observe that bridges in the network (determined by computing biconnected components) are likely to have high edge centrality scores. We apply this heuristic as an optional step (step 1 of Algorithm 10) to determine a set of potential high-centrality edges in the graph, and to accelerate approximate betweenness computation.

4.2.2 Modularity-maximizing agglomerative clustering algorithm (pMA)

A greedy agglomerative approach starts from a state of n singleton communities, and iteratively merges the pair of communities that result in the greatest increase in modularity. Clauset *et al.* [45] give an algorithm that runs in $O(md \log n)$ time, where d is the depth of the resulting dendrogram. The primary data structure is an implicitly-maintained sparse matrix ΔQ, where $\Delta Q(i, j)$ corresponds to a increase in modularity on merging clusters C_i and C_j. We design a new parallel algorithm (pMA, see Algorithm 11) that performs

Algorithm 11: Modularity-maximizing agglomerative clustering algorithm (pMA).

Input: $G(V, E)$, length function $l : E \to \mathbb{R}$
Output: A partition $C = (C_1, ..., C_k)$ ($C_i \neq \phi$ and $C_i \cap C_j = \phi$) of V that maximizes modularity.

```
1  nC ⟵ n;
2  Max heap H ⟵ φ;
3  foreach v ∈ V do
4  |  ΔQd[v] (dynamic array) ⟵ φ;
5  |  ΔQb[v] (multilevel bucket) ⟵ φ;
6  |  Add modularity update value corresponding to each neighbor (adjacent
   |  community) of v to both ΔQb[v] and ΔQd[v].
7  |  Add community-pair with the maximum modularity update value to H.
8  while nC > 1 do
9  |  Select the community pair (i, j) corresponding to the largest value in H.
10 |  Update ΔQd, ΔQb, H in parallel, and increment modularity score.
11 |  nC ⟵ nC − 1;
12 Inspect Q, set C to the clustering with the highest modularity score.
```

the same greedy optimization as Clauset *et al.*'s approach, but uses data representations supported in SNAP for the modularity update matrix. We store each row of the matrix as a sorted dynamic array (so that elements can be identified or inserted in $O(\log n)$ time), as well as a multi-level bucket (to identify the largest element quickly). We parallelize two steps in every iteration of the greedy approach – the matrix rows representing the two communities C_i and C_j are merged in parallel; secondly, if C_i and C_j are connected to other communities, the corresponding ΔQ updates can be parallelized. This algorithm is significantly faster than the divisive clustering approach, with the trade-off of loss in clustering quality for some graph instances.

4.2.3 Greedy local aggregation algorithm (pLA)

Note that the above approaches rely on global metrics for community identification, and parallelism can only be exploited at a very fine granularity (at the level of an iteration). We consider relaxing this further and design an agglomerative partitioning heuristic in which multiple execution threads concurrently try to identify communities. The algorithm proceeds as follows. We first compute biconnected components to determine if the graph has any bridges. If it does, we remove bridge edges and run the connected components kernel.

Algorithm 12: Greedy local aggregation algorithm (pLA).

Input: $G(V, E)$, length function $l : E \to \mathbb{R}$
Output: A partition $C = (C_1, ..., C_k)$ ($C_i \neq \phi$ and $C_i \cap C_j = \phi$) of V that maximizes modularity.

```
1 Run biconnected components to identify bridges.
2 Delete bridges, run connected components.
3 foreach connected component C in G do
4     n_C ⟵ number of vertices in the component;
5     while n_C > 1 do
6         Select a vertex v at random in parallel.
7         Merge vertices/clusters adjacent to v and create a new cluster, based on an
          appropriate local clustering metric (e.g., degree, clustering coefficient).
8         Accept the new cluster if the overall modularity score increases.
9         Update the value of n_C, the number of remaining vertices in the graph.
```

If this splits the graph into multiple isolated components, we run a greedy agglomerative clustering heuristic on each of these components, and finally amalgamate the clusters at the top level. Note that we still optimize for modularity. However, while doing agglomerative clustering, to avoid global synchronization after each iteration, we use a local measure such as degree or clustering coefficient to decide whether an edge needs to be added to a cluster. To initiate clustering, we need to pick a set of seed vertices – this can be done randomly, or obtained from a breadth-first ordering of the vertices. Vertices are greedily added to the clusters, and we exploit parallelism using the *path-limited search* paradigm discussed in the previous section. In practice, this heuristic performs well for networks with a pronounced community structure, and does not rely on any global centrality metrics.

4.3 Experimental Study

		Modularity Q				
Network	n	GN	pBD	pMA	pLA	Best known
`Karate`	34	0.401	0.397	0.381	0.397	0.431 [32]
`Political books`	105	0.509	0.502	0.498	0.487	0.527 [32]
`Jazz musicians`	198	0.405	0.405	0.439	0.398	0.445 [65]
`Metabolic`	453	0.403	0.402	0.402	0.402	0.435 [146]
`E-mail`	1,133	0.532	0.547	0.494	0.487	0.574 [65]
`Key signing`	10,680	0.816	0.846	0.733	0.794	0.855 [146]

Table 6: Modularity scores achieved using GN, pBD, pMA, and pLA. GN corresponds to the Girvan-Newman edge-betweenness based algorithm. The best known modularity scores are determined either by an exhaustive search, or using non-greedy heuristics.

4.3.1 Results and Analysis

We evaluate the performance of the community identification heuristics on twelve different real-world network instances. In Table 6, we compare values of modularity obtained using our new approaches against the Girvan-Newman (GN) algorithm. We show results for six different networks, all of which have been used in previous studies (please see [146, 65] for sources). We also report the best-known modularity score (higher scores indicate better community structure) for each network, obtained by either an exhaustive search, extremal optimization [65], or a simulated annealing-based technique. It should be noted that the approaches used to obtain the best-known modularity scores are computationally very expensive, and may only be applied to small networks. Table 6 shows that our divisive betweenness-based approach pBD performs extremely well in practice, and the modularity scores are comparable to GN. In fact, for the larger *E-mail* and *Key signing* networks, pBD outperforms GN. pMA and pLA, the faster agglomerative algorithms also perform favorably, with pLA giving a better partitioning for the *Karate* and *Key signing* networks.

Label	Network	n	m	Type
`PPI`	human protein interaction network	8,503	32,191	undirected
`Citations`	Citation network from KDD Cup 2003	27,400	352,504	directed
`DBLP`	CS publication coauthorship network	310,138	1,024,262	undirected
`NDwww`	web-crawl (`nd.edu`)	325,729	1,090,107	directed
`Actor`	IMDB movie-actor network	392,400	31,788,592	undirected
`RMAT-SF`	synthetic small-world network	400,000	1,600,000	undirected

Table 7: Networks used in the community identification experimental study.

The real benefit of our algorithms lies in the fact that they are significantly faster than existing approaches, and facilitate analysis of networks that were considered too large to be tractable. We now report execution time and parallel speedup on a multicore parallel system for several real-world graph instances. Table 7 lists a collection of small-world networks gathered from diverse application domains: a protein-interaction network from computational biology, a citation network, a web crawl, and two social networks. We ignore edge directivity in the community detection algorithms. Our test platform for reporting parallel performance results in this paper is the Sun Fire T2000 server, with the Sun UltraSPARC T1 (Niagara) processor. This system has eight cores running at 1.0 GHz, each of

which is four-way multithreaded. The cores share a 3 MB L2 cache, and the system has a main memory of 16 GB. We compile SNAP with the Sun C compiler v5.8 and the default optimization flags.

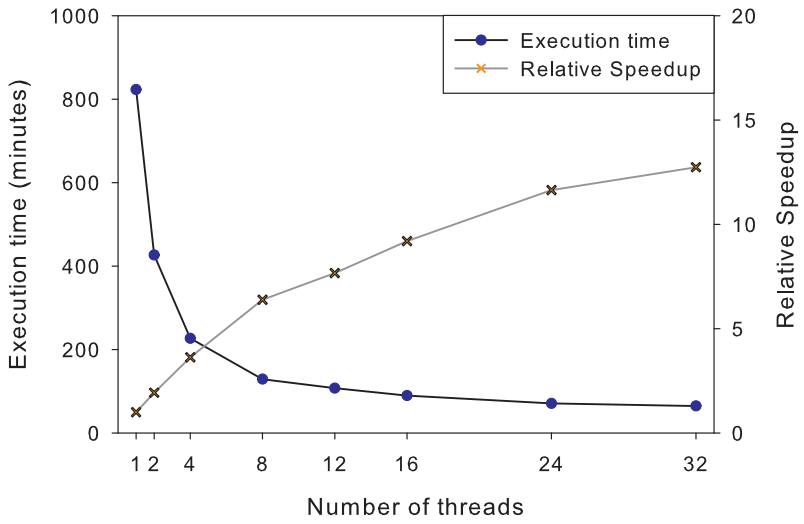


Figure 49: Parallel performance (execution time and relative speedup on the Sun Fire T2000) of the pBD community detection algorithms, when applied to the `RMAT-SF` graph instance.

Figures 49, 51, and 50 give the execution time and relative speedup on the Sun Fire T2000, for community identification using our three parallel algorithms. The graph instance analyzed is *RMAT-SF*, a synthetic small-world network of 0.4 million vertices and 1.6 million edges. The computationally-expensive divisive approximate betweenness approach is the slowest among the three (note that the execution time in Figure 52 is in the order of minutes), while pMA and pLA are comparable in execution time. On 32 threads, we achieve a parallel speedup of roughly 13, 9 and 12 for pBD, pMA, and pLA respectively. These performance results are along expected lines and follow the speedup trends displayed by the SNAP inner kernels such as graph traversal and approximate betweenness centrality [14].

In Figure 52, we compare the performance of pBD to the GN approach for the real-world networks listed in Table 7. pBD is faster than GN because of algorithmic differences

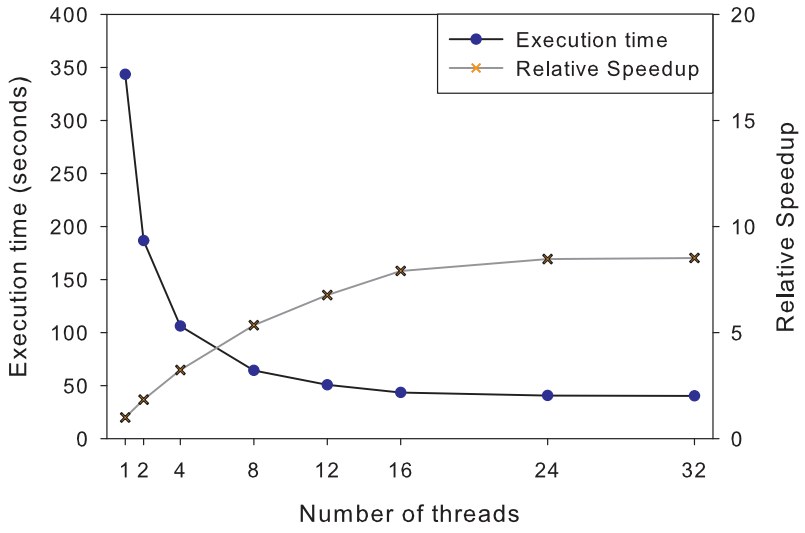


Figure 50: Parallel performance (execution time and relative speedup on the Sun Fire T2000) of the pMA community detection algorithms, when applied to the `RMAT-SF` graph instance.

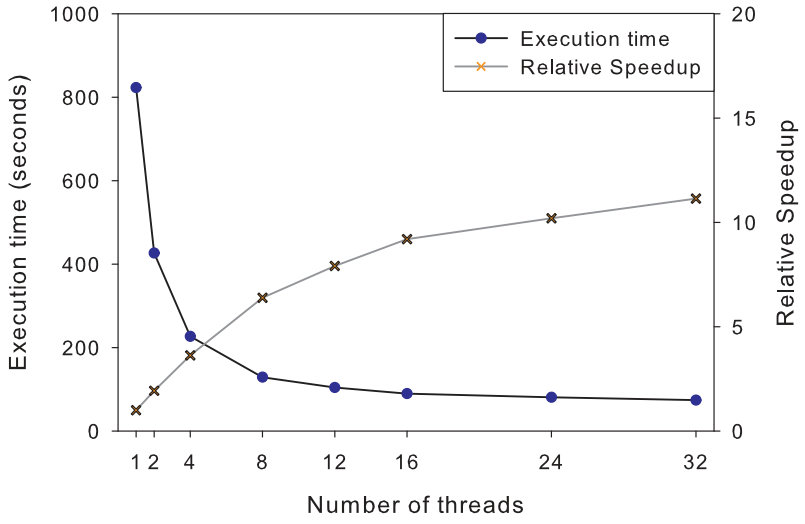


Figure 51: Parallel performance (execution time and relative speedup on the Sun Fire T2000) of the pLA community detection algorithms, when applied to the `RMAT-SF` graph instance.

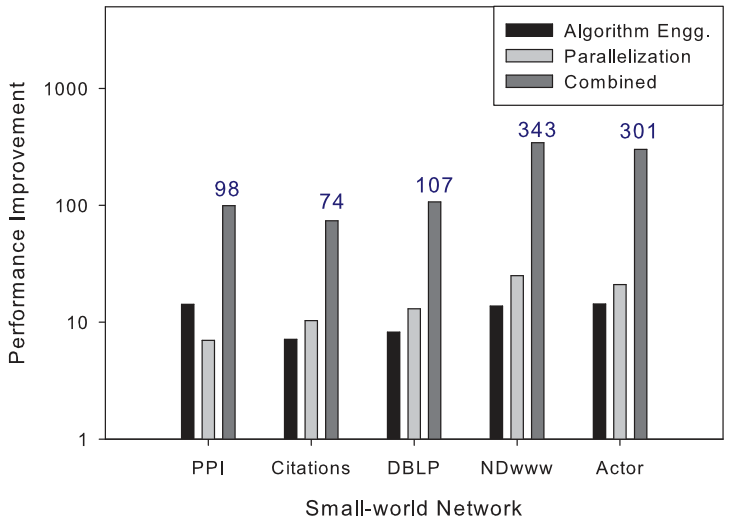


Figure 52: Community Identification: Speedup achieved by pBD over the GN algorithm due to algorithm engineering and parallelization. The bar labels indicates the ratio of execution time of GN to the running time of pBD.

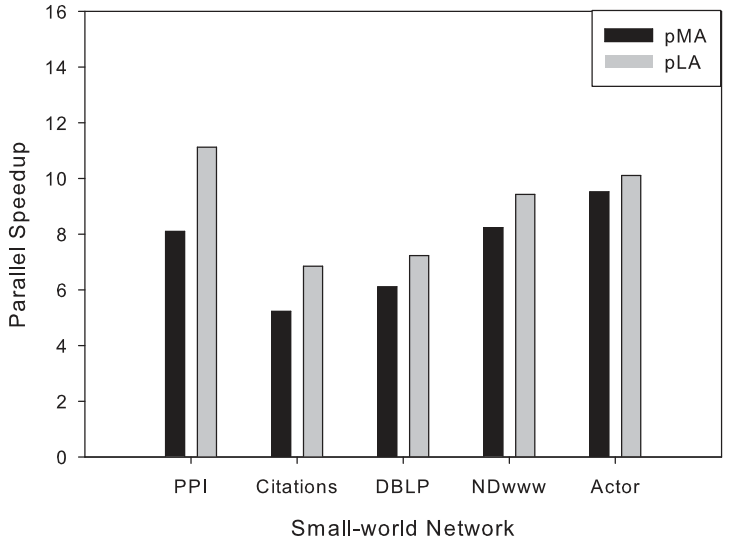


Figure 53: Community Identification: Parallel speedup achieved by pMA and pLA on the test network instances.

(we compute approximate betweenness and incorporate additional small-world network optimizations to speed up the partitioning), and also due to the fact that that it is a parallel approach. Since these speedup factors are multiplicative, the overall performance improvement achieved is quite significant. For instance, for the web-crawl *NDwww*, the single-threaded run of pBD is nearly 26 times faster than an optimized implementation of GN using SNAP. This improvement, coupled with a parallel speedup of 13.2 on the Sun Fire T2000, results in an overall speedup of 343. The performance is consistently high across all the real-world networks. Note that the exact *algorithm engineering* speedup achieved depends on the topology of the network: it is comparatively lower for *PPI* as the network is small. For pMA and pLA, since we do not have a baseline heuristic to compare performance against, we just report the relative speedup on 32 threads for the different graph instances. pLA achieves a slightly higher speedup in most cases, while the running times are comparable.

Note that all three parallel algorithms require only $O(m + n)$-space, independent of the number of processors. While we report performance results for graphs with several millions of vertices and edges in this paper, the algorithms are scalable and can process graphs with even billions of entities.

4.4 SNAP: Small-world Network Analysis and Partitioning Framework

4.4.1 Data Representation

Efficient data structures and graph representations are key to high performance parallel graph algorithms. In order to process massive graphs, it is particularly important that the data structures are space-efficient. The primary graph representation supported in SNAP is a vertex adjacency list representation, implemented using cache-friendly adjacency arrays. This representation is simple and the preferred choice for static graph algorithms. However, for algorithms that require dynamic structural updates to the graph, we need to efficiently support insertion and deletion of edges. We use an auxiliary graph representation that uses dynamic, resizable adjacency arrays. To speed up deletions, adjacencies can be ordered by sorting them by their vertex or edge identifier. Further, several optimizations are possible for

small-world graphs. Small-world networks typically have an unbalanced degree distribution – the majority of the vertices are low-degree ones, and there are a few vertices of very high degree. In such cases, we could have a threshold on the degree and represent low-degree vertex adjacencies in a simple, unsorted adjacency representation, but adjacencies of high-degree vertices in a tree structure such as treaps [166]. Treaps are randomized search trees that support fast insertion, deletion, searching, joining and splitting. In addition, there are efficient parallel algorithms for set operations on treaps such as union, intersection and difference. Based on the graph update rate, and the insertion to deletion ratio for an application, we could choose an appropriate graph representation.

4.4.2 Graph Kernels

The SNAP graph kernels are primarily designed to exploit fine-grained thread level parallelism in graph traversal. We apply one of the following two paradigms in the design of parallel kernels: *level-synchronous* graph traversal, where vertices at each level are visited in parallel; or *path-limited searches*, in which multiple searches are concurrently executed and aggregated. The level-synchronous approach is particularly suitable for small-world networks due to their low graph diameter. Support for fine-grained efficient synchronization is critical in both these approaches. We try to aggressively reduce locking and barrier constructs through algorithmic changes, as well as implementation optimizations. For the BFS kernel, we use a lock-free level-synchronous algorithm that significantly reduces shared memory contention. The minimum spanning tree algorithm uses a lazy synchronization scheme coupled with work-stealing graph traversal to yield a greater granularity of parallelism. While designing fine-grained algorithms for small-world networks, we also consider the unbalanced degree distributions. In a level-synchronized parallel BFS where vertices are statically assigned to processors without considering their degree, it is highly probable that there will be phases with severe work imbalance. To avoid this, we first estimate the processing work to be done from each vertex, and then assign them accordingly to processors. We visit adjacencies of high degree vertices in parallel for better load balancing. With these optimizations, we demonstrate that the performance of our fine-grained BFS and shortest

path algorithms [14, 134] is mostly independent of the graph degree distribution.

We utilize these efficient kernel implementations as building blocks for higher level algorithms such as centrality and partitioning. For these algorithms, we also consider performance trade-offs associated with memory utilization and parallelization granularity. In cases where the input graph instance is small enough, we can trade off space with a coarse-grained parallelization strategy, thus reducing synchronization overhead. We utilize this technique in the compute-intensive ($O(mn)$ work) exact betweenness centrality calculation, where the centrality score computation requires n graph traversals. The fine-grained implementation parallelizing each graph traversal requires $O(m + n)$ space, whereas the memory requirements of the coarse-grained approach, in which the n traversals are distributed among p processors, are $O(p(m + n))$. We also incorporate small-world network specific optimizations in the choice of data structures for centrality computations. For instance, the predecessor sets of a vertex in shortest path computations, required in centrality computations, are implemented differently for low-degree and high-degree vertices. The parallel algorithms, coupled with small-world network optimizations, enable SNAP to analyze networks that are three orders of magnitude larger than the ones that can be processed using commercial and research software packages for SNA (e.g., Pajek [25], InFlow, UCINET).

4.4.3 Network Analysis Metrics and Preprocessing Routines

Most of these metrics have a linear, or sub-linear computational complexity and are straightforward to implement. When used appropriately, they not only provide insight into the network structure, but also help speed up subsequent analysis algorithms. For instance, the average neighbor connectivity metric is a weighted average that gives the average neighbor degree of a degree-k vertex. It is an indicator of whether vertices of a given degree preferentially connect to high- or low-degree vertices. Assortativity coefficient is a related metric proposed by Newman, which is an indicator of community structure in a network. Based on the these metrics, it is easy to identify instances of specific graph classes, such as bipartite graphs, and networks with pronounced community structure. This helps us choose an appropriate community detection algorithm and a clustering measure for which

to optimize. Other preprocessing kernels include computation of connected and biconnected components of the graph. If a graph is composed of several large connected components, it can be decomposed and individual components can be analyzed concurrently. In case of protein interaction networks in computational biology, we find that vertices that are articulation points (determined from computing biconnected components), but have a low degree, are unlikely to be essential to the network [16]. All these preprocessing steps combined together potentially offer an order of magnitude speedup or more [15] for key analysis kernels on real-world network instances.

4.5 dSNAP: Analyzing Dynamic Interaction Networks

In this Section, we demonstrate that the combination of SNA research, dynamic graph algorithms, and high performance computing (HPC) techniques, make massive dynamic interaction graph analysis tractable on current computing systems.

The data stream model [143] is a powerful abstraction for the statistical mining of massive data sets. The key assumptions in this model are that the input data stream may be potentially unbounded and transient, the compute resources are sub-linear in data size, and queries may be answered with only one (or a few) pass(es) over the input data. The bounded memory and computing-resource assumption makes it infeasible to answer most streaming queries exactly, and so approximate answers are acceptable. Effective techniques for approximate query processing include sampling, batch-processing, sketching, and synopsis data structures.

Complementing data stream algorithms, a graph or network representation is a convenient abstraction in many applications: unique data entities are represented as vertices, and the interactions between them are depicted as edges. The vertices and edges can further be typed, classified, or assigned attributes based on relational information from the heterogeneous sources. Analyzing topological characteristics of the network, such as the vertex degree distribution, centrality and community structure, provides valuable insight into the structure and function of the interacting data entities. Common analysis queries on the data set are naturally expressed as variants of problems related to graph connectivity, flow,

or partitioning.

Since classical graph problems are typically formulated in an imperative, state-based manner, it is hard to adapt existing algorithms to the data stream model. New approaches have been proposed for solving graph algorithms in the streaming model [97], but there are no known algorithms to solve fundamental graph problems in sub-linear space and a constant number of data stream passes. The semi-streaming model is a relaxation to the classical streaming model, that allows $O(n\ polylog\ n)$ space and multiple passes over data. This is a simpler model for solving graph problems and several recent successes have been reported [56, 70]. However, this work is far from complete; we require faster exact and approximate algorithms to process peta- and exa-scale data sets, and experimental evaluations of proposed semi-streaming algorithms on current architectures.

While the focus of streaming algorithms is on processing massive amounts of data assuming limited compute and memory resources, the research area of dynamic graph algorithms [59] in graph theory deals with work-efficient algorithms for temporal graph problems. The objective of dynamic graph algorithms is to efficiently maintain a desired graph property (for instance, connectivity, or the spanning tree) under a dynamic setting, i.e. allowing periodic insertion and deletion of edges, and edge weight updates. A dynamic graph algorithm should process queries related to a graph property faster than recomputing from scratch, and also perform topological updates quickly. A *fully dynamic* algorithm handles both edge insertions and deletions, whereas a *partially dynamic* algorithm handles only one of them. Dynamic graph algorithms have been designed for the all-pairs shortest paths, maximum flow, minimum spanning forests and other graph applications [55]. The dynamic tree problem is a key kernel [184] in several dynamic graph algorithms; Eppstein *et al.*'s sparsification [66] and Henzinger *et al.*'s randomization methods [96] are novel algorithmic techniques proposed for processing temporal graphs. Recent experimental studies [188] have evaluated the performance trade-offs involved in some of these kernels and techniques.

4.5.1 Dynamic Network Representation

We model dynamic interaction data by augmenting a static graph G with explicit time-ordering on its edges. In addition, vertices can also store attributes with temporal information, if required. Formally, we model dynamic networks as defined by Kempe *et al.* [116]: a temporal network is a graph $G(V, E)$ where each edge $e \in E$ has a *time-stamp* or *time label* $\lambda(e)$, a non-negative integer value associated with it. This time-stamp may vary depending on the application domain and context: for instance, it can represent the time when the edge was added to the network in some cases, or the time when two entities last interacted in others. If necessary, we can define multiple time labels per edge. We can similarly define time labels $\xi(v)$ for vertices $v \in V$, capturing, for instance, the time when the entity was added. It is straightforward to extend a static graph representation to implement time-stamps defined in the above manner. Also, the time stamps can be abstract entities in the implementation, so that they can be used according to the application requirement.

Efficient data structures and representations are key to high performance dynamic graph algorithms. In order to process massive graphs, it is particularly important that the data structures are space-efficient (compact). Ease of parallelization and the synchronization overhead also influence our representation choice. Ideally, we would like to use simple, scalable and low-overhead (for instance, lock-free) data structures. Finally, the underlying algorithms and applications dictate our data structure choices.

For representing sparse static graphs in SNAP, we used adjacency lists implemented using cache-friendly adjacency arrays [152]. To support insertions and deletions in dynamic graph problems, we can extend the static graph representation and use dynamic, resizable adjacency arrays. Edge deletions are however expensive in this representation, as we may have to scan the entire adjacency list in the worst case to locate the required edge. To speed up deletions, adjacencies can be ordered by sorting them by their vertex or edge identifier. Given the graph update rate and the insertion to deletion ratio for an application, we could choose an appropriate graph representation.

Further, several optimizations are possible for small-world graphs which typically have an unbalanced degree distribution: the majority of the vertices are low-degree ones, and

there are a few vertices of very high degree. In such cases, we could use a threshold on the degree and represent low-degree vertex adjacencies in a simple, unsorted adjacency representation, but adjacencies of high-degree vertices in a tree structure such as treaps [166]. Treaps are randomized search trees that support fast insertion, deletion, searching, joining and splitting. In addition, there are efficient parallel algorithms for set operations on treaps such as union, intersection and difference. Set operations are particularly useful to implement kernels such as graph traversal and induced sub-graphs, and for batch-processing updates.

We conduct an experimental study to assess the performance of various data representations, for different combinations of edge insertions and deletions. We consider three different representations discussed above: dynamic adjacency arrays, sorted dynamic adjacency arrays, and adjacency arrays with treaps. The experiments are conducted on a synthetic graph generated using the R-MAT small-world graph generator [41] for Kronecker graphs. We first generate a directed graph with 2^{18} vertices and 2^{21} edges. We then conduct three different update experiments: 2^{19} insertions and no deletions, $3 * 2^{17}$ insertions and 2^{17} deletions, and 2^{18} insertions and 2^{18} deletions. In Figure 54, we plot the average time taken per update for each of the three data representations on a 3.2 GHz Intel Xeon processor (4 GB memory, 2 MB L2 cache). Observe that the dynamic adjacency array representation is fastest for cases when the majority of updates are insertions. Treaps are slower than sorted dynamic arrays, but only by a constant factor. They give the same average update time, regardless of the sequence and composition of the updates.

Figure 55 plots the parallel performance (average time per structural update and relative speedup) for executing a sequence of 2^{22} insertions and 2^{20} on the Sun Fire T2000 server with the UltraSparc T1 Niagara processor (eight cores at 1.0 GHz, each of which is four-way multithreaded; 3 MB shared L2 cache, main memory of 16 GB). The input graph is a synthetic small-world undirected network with 32 million vertices and 128 million edges, and we use a treap-adjacency array representation. Observe that the structural updates are efficiently processed in parallel, and we achieve a speedup of close to 12 on 32 threads of the Sun Fire T2000 system.

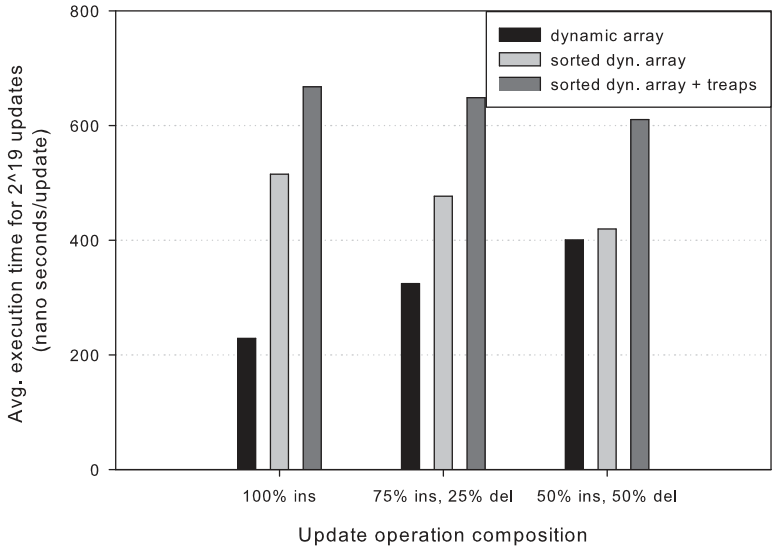


Figure 54: Dynamic networks: Time taken per structural update for various temporal graph representations (synthetic scale-free graph of 2^{18} vertices and 2^{21} edges).

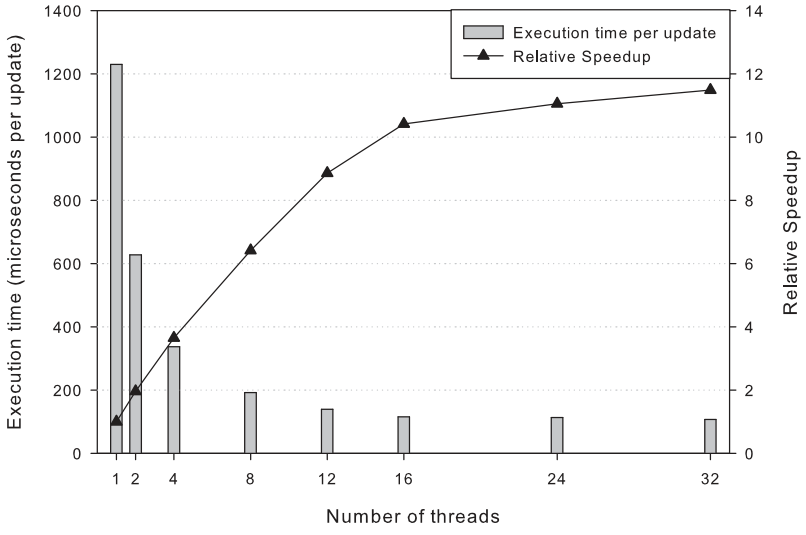


Figure 55: Performance of structural update operations on the Sun Fire T2000 system.

Compressed graph structures are another attractive design choice for processing massive networks, and they have been extensively studied in the context of web-graphs [29]. Exploiting the small-world and self-similarity properties of the web-graph, mechanisms such as vertex reordering, compact interval representations, and compression of similar adjacency lists have been proposed. It is an open question on how these techniques perform for real-world networks from other applications, and whether they can be extended for processing dynamic graphs.

Maintaining a forest that changes over time with edge insertions and deletions, known as the dynamic tree problem, is a key kernel in dynamic graph and network flow problems [184]. To support logarithmic-time updates and tree operations, several data structures have been proposed (for example, the dynamic trees of Sleator and Tarjan (ST-trees), top trees, and RC-trees). These data structures primarily partition the tree into a set of vertex-disjoint paths and maintain it under a series of updates. ST-trees represent each path using a binary tree and are easy to implement, but cannot be easily adapted to other applications. Topology trees use a sophisticated partition based on the topology of the trees, but simpler data structures to represent paths. Topology trees, top trees and RC-trees are based on tree contraction, and progressively combine vertices to obtain a hierarchical representation of the tree. A recent experimental study [172] shows that a linear-time implementation of ST-trees performs extremely well for low-diameter graphs. Since small-world graphs have a low diameter, this data structure is attractive for dynamic graph problems.

4.5.2 Dynamic Graph Kernels

We next identify key *kernels* (or *algorithmic building blocks* in the design of higher-level analysis approaches) for studying temporal networks, and present new parallel algorithms for solving them.

Induced Subgraphs. Utilizing temporal information, several dynamic graph problems can be reformulated as problems on static instances. Given edge and vertex time labels, it is straight-forward to extract and analyze snapshots of a network. For instance, centrality and connectivity queries on a dynamic network at a particular time instant, or queries on

entities added in a specified time interval, can be solved by first extracting the induced subgraph. This approach is feasible on HPC systems when the graph fits in the main memory. Thus, an efficient implementation of the induced subgraph kernel would facilitate solving several dynamic graph problems. Fortunately, the induced subgraph kernel is easy to parallelize, and is very similar to original graph generation, with insertion and deletions of edges. The execution time is dependent on the size and connectivity of the vertex or edge set that induce the subgraph.

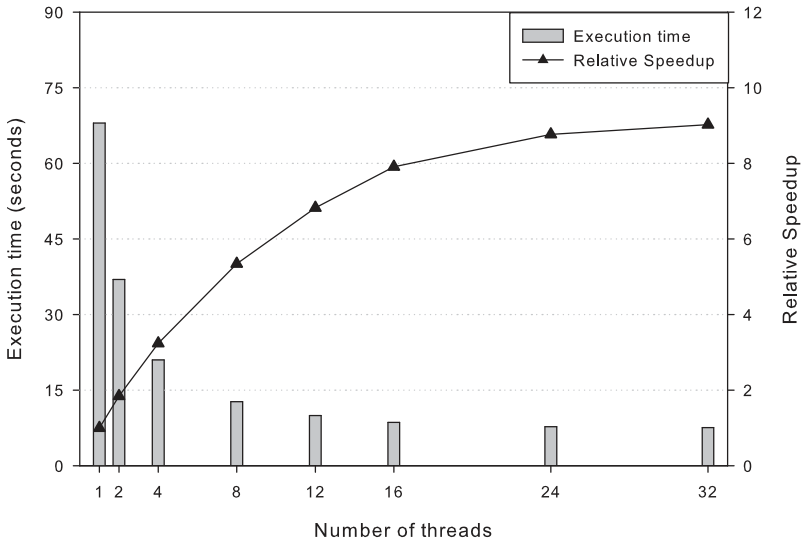


Figure 56: Performance of induced subgraphs on the Sun Fire T2000 system.

Figure 56 plots the parallel performance of the induced subgraph kernel on a Sun Fire T2000. We apply the induced subgraph kernel on a graph of 30 million vertices and 200 million edges, with each edge randomly assigned a integral time stamp between 1 and 100. We filter the subgraph corresponding to the edges inserted in time interval (0, 20). The first phase of the algorithm, identification of edges that need to be inserted or deleted, is easily parallelized. The problem then reduces to a series of insertions or deletions to create a data representation for the induced network. Thus, each edge in the graph is visited twice in this kernel. As demonstrated in Figure 56, the induced subgraph kernel achieves an impressive parallel speedup on the Sun Fire T2000 system.

Breadth-first Search. Graph traversal is a fundamental technique used in several

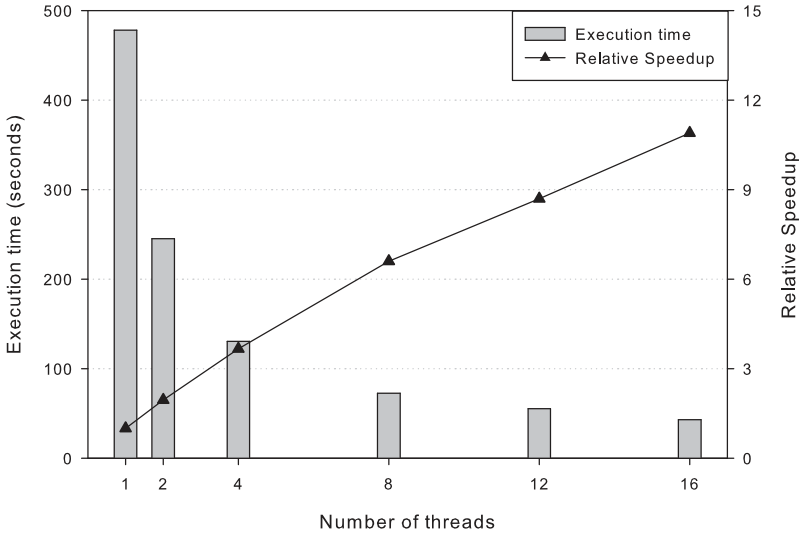


Figure 57: Parallel BFS performance on the IBM Power 570.

network algorithms. Breadth-first search (BFS) is an important approach for the systematic exploration of large-scale networks. In prior work, we designed a level-synchronous PRAM algorithm for BFS that takes $O(d)$ time and optimal linear work (d is the graph diameter) [14]. For small-world graphs, where d is typically a constant, or in some cases $O(\log n)$, this algorithm can be efficiently implemented on large shared memory parallel systems with a high parallel speedup.

For isolated runs of BFS on dynamic graphs, we can take the approach discussed in the induced subgraph kernel, i.e., utilize the time-stamp information and recompute from scratch. This approach requires no additional memory, as it just uses the time label information for filtering vertices and edges during graph traversal. Coupled with optimizations to handle graphs with unbalanced degree distributions, we are able to traverse massive graphs in just a few seconds on current HPC systems. For instance, on 16 processors of an IBM Power 570 symmetric multiprocessor system (16 1.9 GHz Power5 cores with SMT, 256 GB shared memory), BFS on a scale-free graph of 500 million vertices and 4 billion edges with time-stamped information takes just 46 seconds (see Figure 57 for parallel performance and scaling).

Shortest paths. The dynamic version of the shortest path problem deals with

handling structural updates on the graph, while answering shortest path queries without recomputing from scratch. Both the single-source (SSSP) and all-pairs versions of this problem have been studied extensively. We are interested in speeding up shortest-path enumeration based centrality metrics such as closeness and betweenness that use SSSP as a kernel, and so we focus on dynamic SSSP in this paper. Ramalingam and Reps [157] propose one of the earliest sequential algorithms for dynamic SSSP that has been found to perform well in practice. Faster randomized algorithms for this problem are reviewed in [55] and [59], but we design a parallel formulation of the Ramalingam-Reps (RR) algorithm for its simplicity, ease of parallelization, and proven practical performance. This algorithm formally presents the idea of an *affected region* in the graph due to a edge deletion. An edge $\langle x, y\rangle$ is said to be affected by the deletion of the edge $\langle v, w\rangle$ if there exists no path in the new graph from x to the sink that makes use of the edge $\langle x, y\rangle$ and has a length equal to the old $d(x)$. It is easily seen that $\langle x, y\rangle$ is an affected edge iff y is an affected vertex. On the other hand, any vertex x other than u (the source of the deleted edge) is an affected vertex iff all edges going out of x are affected edges. The vertex u itself is an affected vertex iff $\langle u, w\rangle$ is the only edge going out of vertex u.

The RR algorithm for updating the shortest path tree after the deletion of an edge works in two phases. The first phase computes the set of all affected vertices and edges and removes the affected edges from the shortest path tree, while the second phase computes the new distance value for all the affected vertices and updates the tree appropriately. The first phase is very similar to a topological ordering algorithm, whereas the second is based on a batched version of Dijkstra's shortest path algorithm. We designed and implemented two parallel algorithms for SSSP [134] that perform very well for low-diameter graphs. Our Δ-stepping implementation can be modified to solve Phase 2 of the RR algorithm. In combination with a parallelization of phase 1 of the algorithm, our dynamic SSSP implementation yields a substantial speedup over the naïve approach of recomputing the SSSP tree from scratch.

Connectivity. Given any two vertices s and t, a connectivity query asks whether there is a path connecting s to t. In a dynamic setting, the connectivity problem reduces to the problem of maintaining a spanning forest in G, i.e., maintaining a spanning tree

for each connected component of G. Dynamic connectivity is studied both in a fully and in a partially dynamic setting. A deterministic algorithm for fully dynamic connectivity achieves $O(\log^2 n)$ update time and $O(\log n / \log\log n)$ query time [55]. We parallelize the operations defined for an ST-tree, which is shown to work well sequentially in practice for low-diameter graphs [172]. Since algorithms with poly-logarithmic update and query times require $O(m + n \log n)$ preprocessing time and space, dynamically maintaining a spanning forest is useful only when the rate of updates and connectivity queries is relatively high. Otherwise, with a brute-force bidirectional BFS, we can determine whether s and t are connected, and the shortest path length connecting them, very quickly in parallel [14]. This approach requires no preprocessing, and if structural updates are infrequent, they can be easily processed by assigning time labels.

We next present case studies of two social network analysis problems that can be parallelized efficiently using the data structures and kernels presented in the previous sections – *centrality analysis* for identifying key entities in an interaction network, and *community identification* for detecting dense substructures in temporal data sets.

4.5.3 Betweenness and Community Identification in Dynamic Networks

Betweenness centrality can be formulated for entities in a dynamic network, by taking the interaction time labels and their ordering into consideration. Define a temporal path $p_d\langle u, v\rangle$ [116] between u and v as a sequence of edges $\langle u = v_0, v_1\rangle$, $\langle v_1, v_2\rangle$, ..., $\langle v_{k-1}, v_k = v\rangle$, such that $\lambda(v_{t-1}, v_t) < \lambda(v_t, v_{t+1})$ for $t = 1, 2, ..., k-1$. We define the *shortest temporal path* between u and v as the temporal path with the shortest distance $d(u, v)$. In this framework, the temporal betweenness centrality $BC_d(v)$ of a vertex v is the sum of the fraction of all shortest temporal paths passing through v, between all pairs of vertices in the graph. Formally, let $\delta_{st}(v)$ denote the pairwise dependency, or the fraction of shortest temporal paths between s and t that pass through v: $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$. Then, $BC_d(v) = \sum_{s \neq v \neq t \in V} \delta_{st}(v)$. Note that a shortest temporal path between u and v may not necessarily be a shortest path in the aggregated graph formed by ignoring time stamps. Our definition of temporal centrality respects the time ordering of edges and provides better insight into

network evolution.

Using data structures from Section 4.5.1 and optimized kernels for dynamic networks, we design and implement a new parallel approach for computing temporal betweenness centrality. We augment our prior parallel algorithm for exact betweenness computation on static graphs [15] with time-stamp information. The graph traversal and shortest path-counting steps in our parallel approach for static graphs are modified to process dynamic networks, but the parallel performance of the dynamic implementation is nearly identical to the static approach.

Algorithm 13: Temporal betweenness centrality-based divisive clustering algorithm.

Input: $G(V, E)$, length function $l : E \rightarrow \mathbb{R}$, time-stamp $\lambda(e) \forall\ e \in E$.
Output: A partition $C = (C_1, ..., C_k)$ ($C_i \neq \phi$ and $C_i \cap C_j = \phi$) of V that maximizes modularity; A dendrogram D representing the clustering steps.

1 Preprocessing step: Compute *Biconnected components*, identify articulation points and bridges.
2 $numIter \longleftarrow 0$;
3 **while** $numIter < m$ **do**
4 Find edge e_m with the highest *approximate temporal betweenness centrality* score **in parallel**.
5 Mark edge e_m as *deleted* in the graph G.
6 Run connected components on G, update dendrogram and number of clusters **in parallel**.
7 Compute modularity of the current partitioning **in parallel**.
8 $numInter \longleftarrow numIter + 1$;
end
9 Inspect the dendrogram, set C to the clustering with the highest modularity score.

Algorithm 13 presents a new parallel approach for determining communities in large dynamic networks. It is based on the idea of approximately computing temporal betweenness centrality iteratively, and gradually removing high centrality edges from the network. We maximize modularity, a measure that is based on optimizing intra-cluster density over inter-cluster sparsity.

4.6 Summary

We discuss the design, implementation, and performance of three novel parallel community detection algorithms that optimize modularity, a popular measure for clustering quality

in social network analysis. In order to achieve scalable parallel performance, we exploit typical network characteristics of small-world networks, such as the low graph diameter, sparse connectivity, and skewed degree distribution. We conduct an extensive experimental study on real-world graph instances and demonstrate that our parallel schemes, coupled with aggressive algorithm engineering for small-world networks, give significant running time improvements over existing modularity-based clustering heuristics, with little or no loss in clustering quality. For instance, our divisive clustering approach based on approximate edge betweenness centrality is *more than two orders of magnitude* faster than a competing greedy approach, for a variety of large graph instances on the Sun Fire T2000 multicore system.

We also present high-performance combinatorial techniques for analyzing large-scale information networks, encapsulating dynamic interaction data in the order of billions of entities. For tractable analysis of massive temporal data sets, we need holistic techniques that supplement existing approaches for processing static graphs with relevant ideas from dynamic graph algorithms, social network analysis, and parallel algorithms for combinatorial problems. For instance, in order to design scalable parallel algorithms, it is crucial to estimate and exploit topological characteristics such as the degree of data clustering and connectivity. We present new techniques for the systematic analysis of dynamic networks: we experiment with several graph representations, design and implement key graph analysis kernels, and present new parallel algorithms for analyzing large-scale dynamic graphs. The source code is freely available as the dSNAP module in our open-source SNAP (Small-world Network Analysis and Partitioning) graph framework from sourceforge.

CHAPTER V

CONCLUSIONS

We present SNAP, a new framework for analyzing massive complex networks. We discuss new parallel approaches underlying the SNAP framework, and present the design and implementation of algorithms for large-scale graph traversal, shortest paths, centrality, and community identification.

We demonstrate that the massive multithreading paradigm of the Cray MTA-2 aids in the design of simple, scalable and high-performance graph algorithms. We achieve impressive results for BFS and *st*-connectivity on the MTA-2 for several large-scale real and synthetic graph instances, both for algorithm execution time and parallel performance. For instance, BFS on a scale-free graph of 200 million vertices and 1 billion edges takes less than 5 seconds on a 40-processor MTA-2 system, with an absolute speedup of close to 30. We also achieve parallel speedup on the multicore Sun Niagara and the IBM p5 570 SMP, for a variety of graph instances. These are significant results in parallel computing, as prior implementations of graph algorithms report very limited or no speedup on irregular and sparse graphs, when compared to their best sequential implementations. The absolute execution time values are significant; linear-work problems involving large graphs with billions of vertices and edges can be solved in seconds on current HPC systems.

We present new parallel algorithms for the single source shortest paths problem, with an extensive experimental evaluation of the parallel Δ-stepping algorithm. We analyze the performance using platform-independent Δ-stepping algorithm operation counts such as the number of *phases*, and the *request set sizes*, to explain performance across graph families. For grids and road networks, we observe that the average request set size is much smaller than corresponding low-diameter graph instances of the same size. Also, the parallelization overhead is significant for these instances, as there are a higher number of parallel phases. We also study the dependence of the bucket-width Δ on the parallel performance of the

algorithm. For high diameter graphs, there is a trade-off between the number of phases and the amount of work done (proportional to the number of bucket insertions). The execution time is dependent on the value of Δ as well as the number of processors. In future, we will extend this study to include optimized implementations of Δ-stepping on symmetric multiprocessors and multicore processors.

Centrality analysis is an important problem in complex network analysis. We present the first parallel algorithms for exactly computing several compute-intensive centrality measures on real-world networks. Our parallel algorithms exploit common topological properties of small-world networks such as the low diameter and the unbalanced degree distributions. We compute exact betweenness centrality for several large networks such as web crawls, protein-interaction networks, movie-actor and patent citation networks. These graph instances are *three orders of magnitude larger* than the problem sizes that can be processed by current social network analysis packages. We also achieve impressive parallel performance on several multicore, symmetric multiprocessor and multithreaded systems. For instance, we compute the exact betweenness centrality value for each vertex in a large US patent citation network (3 million patents, 16 million citations) in 42 minutes on 16 processors, utilizing 20GB RAM of the IBM Power 570 SMP system. Current network analysis packages on the other hand cannot process graphs with more than hundred thousand edges.

We also present a novel approximation algorithm for computing betweenness centrality of a given vertex, in both weighted and unweighted graphs. Our approximation algorithm is based on an adaptive sampling technique that significantly reduces the number of single-source shortest path computations for vertices with high centrality. We conduct an extensive experimental study on real-world graph instances, and observe that the approximation algorithm performs well on web crawls, road networks and biological networks. Approximating the centrality of *all* vertices in time less than $O(nm)$ for unweighted graphs and $O(nm + n^2 \log n)$ for weighted graphs is an open problem. Also, designing a fully dynamic algorithm for computing betweenness is a challenging research problem.

As a case study, we conduct an extensive analysis of the global topological characteristics of the human protein interaction network. We report a new topology feature in the yeast and

human PINs not found in synthetic scale-free networks: the prevalence of low degree proteins with high-betweenness values. Our results show that existing evolutionary algorithms that produce scale-free networks do not predict the existence of high-betweenness, low-degree vertices found within the yeast and human PINs. The high-betweenness, low centrality vertices also provide some insight into the clustering nature and coreness of the network. We find that vertices with high centrality scores are very likely to be articulation points in the graph, and also have low clustering coefficients. The source code for the various graph analysis routines is freely available online from our web site. We also intend to provide our centrality analysis codes as plug-ins to biological network visualization tools.

We present the design, implementation, and performance analysis of three novel parallel community detection algorithms that optimize modularity, a popular measure for clustering quality in social network analysis. We conduct an extensive experimental study on real-world graph instances and demonstrate that our parallel schemes, coupled with aggressive algorithm engineering for small-world networks, give significant running time improvements over existing modularity-based clustering heuristics, with little or no loss in clustering quality. For instance, our divisive clustering approach based on approximate edge betweenness centrality is *more than two orders of magnitude* faster than a competing greedy approach, for a variety of large graph instances on the Sun Fire T2000 multicore system.

As part of ongoing work, we are designing new algorithms for complex network analysis kernels and incorporating existing techniques into SNAP. Our current focus is on support for spectral analysis techniques, and efficient parallel implementations of spectral algorithms that optimize modularity.

The area of dynamic interaction network analysis poses several research challenges. We intend to extend SNAP to support extensive analysis of dynamic networks. In this dissertation, we present new graph representations for dynamic networks, and the design and implementation of key graph analysis kernels. However, the formulation and efficient parallel implementation of several complex network analysis routines still remain open problems.

APPENDIX A

Δ-STEPPING ALGORITHM TABLES

A.1 Sequential performance of Δ-stepping implementation on the reference platform

Table 8: Sequential performance (execution time in seconds, and normalized performance with reference to the baseline BFS) of our Δ-stepping implementation for the core random graph families.

(a) Random4-n core family. Problem instance denotes the log of the number of vertices. A directed random graph of n vertices, $m = 4n$ edges, and maximum weight $C = n$.

Problem Instance	**11**	**12**	**13**	**14**	**15**	**16**
BFS	.0001	.0003	.0006	.001	.004	.02
Δ-stepping	.0007	.002	.004	.01	.03	.09
Normalized to BFS	7.00	6.67	6.67	10.00	7.50	4.50
DIMACS Reference	.0003	.0008	.002	.008	.02	.06
Normalized to BFS	3.00	2.67	3.33	8.00	5.00	3.00

Problem Instance	**17**	**18**	**19**	**20**	**21**
BFS	.05	.14	.32	.69	1.45
Δ-stepping	.23	.52	1.12	2.54	5.42
Normalized to BFS	4.60	3.71	3.50	3.68	3.74
DIMACS Reference	.13	.30	0.65	1.39	3.19
Normalized to BFS	2.60	2.14	2.03	2.01	2.20

(b) Random4-C core family. Problem instance denotes the log of the maximum edge weight. $n = 2^{20}$ vertices and $m = 4n$ edges.

Problem Instance	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
BFS	0.69	0.69	0.69	0.69	0.69	0.69	0.69	0.69
Δ-stepping	2.31	2.55	2.53	2.55	2.54	2.54	2.54	2.54
Normalized to BFS	3.35	3.70	3.67	3.70	3.68	3.67	3.67	3.67
DIMACS Reference	0.87	0.89	0.92	1.21	1.26	1.31	1.38	1.36
Normalized to BFS	1.26	1.29	1.33	1.75	1.83	1.90	2.00	1.97

Problem Instance	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
BFS	0.69	0.69	0.69	0.69	0.69	0.69	0.69	0.69
Δ-stepping	2.55	2.54	2.54	2.55	2.54	2.54	2.54	2.54
Normalized to BFS	3.70	3.68	3.68	3.70	3.68	3.68	3.68	3.68
DIMACS Reference	1.37	1.37	1.38	1.37	1.37	1.38	1.37	1.38
Normalized to BFS	1.98	1.98	2.00	1.98	1.98	2.00	1.98	2.00

Table 9: Sequential performance (execution time in seconds, and normalized performance with reference to the baseline BFS) of our Δ-stepping implementation for the core long grid graph families.

(a) Long-n core family. Problem instance i denotes a grid with $x = 2^i$ and $y = 16$. $n = xy$ and $\frac{m}{n} \approx 4$.

Problem Instance	**6**	**7**	**8**	**9**	**10**	**11**
BFS	.0001	.0002	.0003	.0007	.001	.004
Δ-stepping	.0005	.001	.002	.005	.01	.03
Normalized to BFS	5.00	5.00	6.67	7.14	10.00	7.50
DIMACS Reference	.0002	.0003	.0007	.002	.006	0.01
Normalized to BFS	2.00	1.50	2.33	2.86	6.00	2.50

Problem Instance	**12**	**13**	**14**	**15**	**16**
BFS	.02	.04	.09	.19	.41
Δ-stepping	.07	.17	.35	.76	1.54
Normalized to BFS	3.50	4.25	3.89	4.00	3.76
DIMACS Reference	.03	0.06	.13	.27	.60
Normalized to BFS	1.50	1.50	1.44	1.42	1.46

(b) Long-C core family. Problem instance denotes the log of the maximum edge weight. The grid dimensions are set to $x = 2^{16}$ and $y = 16$.

Problem Instance	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
BFS	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
Δ-stepping	0.68	0.75	0.78	0.88	1.02	1.07	1.09	1.09
Normalized to BFS	2.75	3.00	3.12	3.52	4.08	4.28	4.36	4.36
DIMACS Reference	0.50	0.54	0.57	0.59	0.57	0.58	0.60	0.60
Normalized to BFS	2.00	2.16	2.28	2.36	2.28	2.32	2.40	2.40

Problem Instance	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
BFS	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
Δ-stepping	1.08	1.09	1.09	1.09	1.08	1.09	1.08	1.09
Normalized to BFS	4.32	4.36	4.36	4.36	4.32	4.36	4.32	4.36
DIMACS Reference	0.59	0.60	0.61	0.59	0.61	0.60	0.60	0.60
Normalized to BFS	2.36	2.40	2.44	2.36	2.44	2.40	2.40	2.40

Table 10: Sequential performance (execution time in seconds, and normalized performance with reference to the baseline BFS) of our Δ-stepping implementation for the core square grid graph families.

(a) Square-n core family. Problem instance denotes the log of the grid x dimension. $x = y$ and $\frac{m}{n} \approx 4$.

Problem Instance	**11**	**12**	**13**	**14**	**15**	**16**
BFS	.0001	.0003	.0007	.001	.003	.01
Δ-stepping	.0008	.002	.004	.008	.03	.07
Normalized to BFS	8.00	6.67	5.71	8.00	10.00	7.00
DIMACS Reference	.0003	.0007	.002	.006	.01	.03
Normalized to BFS	3.00	2.33	2.86	6.00	3.33	3.00

Problem Instance	**17**	**18**	**19**	**20**	**21**
BFS	.04	.08	.20	.42	.93
Δ-stepping	.20	.36	.81	2.05	4.38
Normalized to BFS	5.00	4.00	4.05	4.88	4.71
DIMACS Reference	.06	0.14	.36	.84	2.01
Normalized to BFS	1.50	1.75	1.80	2.00	2.16

(b) Square-C core family. Problem instance denotes the log of the edge weight. The grid dimensions are set to $x = y = 2^{10}$, and $n = xy$.

Problem Instance	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
BFS	0.42	0.42	0.42	0.42	0.42	0.42	0.42	0.42
Δ-stepping	1.99	2.06	2.03	2.09	2.05	2.07	2.06	2.01
Normalized to BFS	4.74	4.90	4.83	4.89	4.88	4.93	4.90	4.79
DIMACS Reference	0.56	0.68	0.71	0.79	0.78	0.76	0.81	0.80
Normalized to BFS	1.33	1.62	1.69	1.88	1.86	1.81	1.93	1.90

Problem Instance	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
BFS	0.42	0.42	0.42	0.42	0.42	0.42	0.42	0.42
Δ-stepping	2.04	2.09	2.05	2.06	2.01	2.08	2.09	2.08
Normalized to BFS	4.86	4.98	4.88	4.90	4.78	4.95	4.98	4.95
DIMACS Reference	0.82	0.80	0.83	0.79	0.77	0.79	0.78	0.77
Normalized to BFS	1.95	1.90	1.98	1.88	1.83	1.88	1.86	1.83

Table 11: Sequential performance (execution time in seconds, and normalized performance with reference to the baseline BFS) of our Δ-stepping implementation for the core road networks.

(a) Core graphs from the USA road network, with the transit time as the length function.

Problem Instance	**CTR**	**W**	**E**	**LKS**	**CAL**	**NE**
BFS	4.16	1.49	.65	.39	.26	.17
Δ-stepping	25.24	9.87	4.97	2.52	1.95	1.43
Normalized to BFS	6.07	6.63	7.65	6.46	7.50	8.41
DIMACS Reference	9.06	3.12	1.65	1.14	.72	.58
Normalized to BFS	2.18	2.09	2.54	2.92	2.77	3.41

Problem Instance	**NW**	**FLA**	**COL**	**BAY**	**NY**
BFS	.16	.13	.04	0.03	.02
Δ-stepping	.89	.97	.30	.21	.15
Normalized to BFS	5.56	7.46	7.5	7.00	7.50
DIMACS Reference	.45	.36	.13	.09	.07
Normalized to BFS	2.81	2.77	3.25	3.00	3.50

(b) Core graphs from the USA road network, with the distance as the length function.

Problem Instance	**CTR**	**W**	**E**	**LKS**	**CAL**	**NE**
BFS	4.32	1.89	1.05	.80	.54	.34
Δ-stepping	21.63	10.34	7.02	3.52	3.67	1.06
Normalized to BFS	5.01	5.47	6.69	4.40	6.80	3.11
DIMACS Reference	15.52	4.91	3.12	2.24	1.41	0.86
Normalized to BFS	3.59	2.60	2.97	2.80	2.61	2.53

Problem Instance	**NW**	**FLA**	**COL**	**BAY**	**NY**
BFS	.31	.28	.05	.03	.02
Δ-stepping	1.26	1.17	0.15	0.11	0.08
Normalized to BFS	4.06	4.18	3.00	3.67	4.00
DIMACS Reference	0.71	0.55	0.13	0.08	0.07
Normalized to BFS	2.29	1.96	2.60	2.67	3.50

A.2 Algorithm performance as a function of Δ

Table 12: Performance of the Δ-stepping algorithm as a function of the bucket width Δ for mesh networks.

(a) Long grid instance ($n = C = 2^{20}$).

Δ	0.1	0.5	1	5	10
No. of phases	295.27K	151.43K	124.76K	97.38K	92.70K
Last non-empty bucket	216.38K	43.28K	21.64K	4.33K	2.16K
Average distance	10805	10805	10805	10805	10805
Avg. no. of light relax requests per phase	0.65	5.49	11.29	22.99	37.22
Avg. no. of heavy relax requests per bucket	5.21	9.65	0	0	0
Total number of relaxations	1.32M	1.25M	1.40M	2.23M	3.45M
Execution Time (40 processors MTA-2, seconds)	858.75	465.14	443.05	369.57	274.23

(b) Square grid instance ($n = C = 2^{20}$).

Δ	0.1	0.5	1	5	10
No. of phases	12795	5489	4188	2769	2504
Last non-empty bucket	4691	938	469	93	46
Average distance	251.86	251.86	251.86	251.86	251.86
Avg. no. of light relax requests per phase	15.51	155.11	340.50	785.18	1248.72
Avg. no. of heavy relax requests per bucket	242.22	437.44	0	0	0
Total number of relaxations	1.33M	1.26M	1.43M	2.17M	3.13M
Execution Time (40 processors MTA-2, seconds)	48.77	20.04	13.92	9.17	8.32

Table 13: Performance of the Δ-stepping algorithm as a function of the bucket width Δ for random and scale-free networks.

(a) Random4-n graph instance ($n = C = 2^{28}$, $m = 4n$).

Δ	0.1	0.5	1	5	10
No. of phases	328	122	89	58	50
Last non-empty bucket	93	19	9	1	0
Average distance	4.94	4.94	4.94	4.94	4.94
Avg. no. of light relax requests per phase	161K	1.84M	4.43M	8.28M	20.00M
Avg. no. of heavy relax requests per bucket	3.12M	5.46M	0	0	0
Total number of relaxations	343.20M	328.70M	394.30M	480.30M	1.00B
Execution Time (40 processors MTA-2, seconds)	14.03	11.64	13.57	16.15	27.14

(b) ScaleFree4-n graph instance ($n = C = 2^{25}$, $m = 4n$).

Δ	0.1	0.5	1	5	10
No. of phases	312	117	83	51	39
Last non-empty bucket	131	26	13	2	1
Average distance	1.68	1.68	1.68	1.68	1.68
Avg. no. of light relax requests per phase	22.40K	267.00K	667.00K	2.40M	3.15M
Avg. no. of heavy relax requests per bucket	278.00K	455.80K	0	0	0
Total number of relaxations	43.78M	43.63M	55.40M	122.68M	122.76M
Execution Time (40 processors MTA-2, seconds)	4.23	2.55	2.79	5.48	6.38

(c) RandomLogUnif4-n instance ($n = C = 2^{20}$, $m = 4n$).

Δ	0.001	0.05	0.1	0.5	1
No. of phases	460	115	93	77	71
Last non-empty bucket	134	17	8	4	2
Average distance	0.04	0.04	0.04	0.04	0.04
Avg. no. of light relax requests per phase	1.50K	50.80K	84.80K	132.00K	150.01K
Avg. no. of heavy relax requests per bucket	6.50K	3.47K	3.97K	2.18K	1.42K
Total number of relaxations	1.59M	5.91M	7.92M	10.17M	10.74M
Execution Time (40 processors MTA-2, seconds)	2.15	1.18	0.96	0.80	0.75

Table 14: Performance of the Δ-stepping algorithm as a function of the bucket width Δ for road networks.

(a) Central USA road instance (distance).

Δ	0.1	0.5	1	5
No. of phases	3129	2017	1669	1300
Last non-empty bucket	105	21	10	2
Average distance	3.93	3.93	3.93	3.93
Avg. no. of light relax requests per phase	6.34K	26.04K	57.89K	82.60K
Avg. no. of heavy relax requests per bucket	320.60	1.27	0	0
Total number of relaxations	19.87M	52.50M	96.60M	107.00M
Execution Time (40 processors MTA-2, seconds)	7.83	5.84	5.62	8.88

(b) NE USA road instance (transit time).

Δ	0.1	0.5	1	5
No. of phases	437	3542	3126	2220
Last non-empty bucket	315	63	31	6
Average distance	14.06	14.06	14.06	14.06
Avg. no. of light relax requests per phase	369.90	783.80	1.38K	8.66K
Avg. no. of heavy relax requests per bucket	168.40	1.85	0	0
Total number of relaxations	1.76M	2.78M	4.31M	19.21M
Execution Time (40 processors MTA-2, seconds)	12.92	9.25	8.39	6.84

A.3 Parallel Performance on the Cray MTA-2

Table 15: MTA-2 parallel performance (execution time in seconds, normalized performance with reference to the baseline BFS, relative speedup) of our Δ-stepping implementation on Random4-n graphs. Problem instance denotes log of the number of vertices. p denotes the number of processors. $m = 4n$ edges, and maximum weight $C = n$.

p	Problem Instance	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**
1	BFS (sec)	0.62	1.24	3.39	4.91	9.70	18.90	37.30	73.94
	Δ-stepping (sec)	3.21	6.34	12.05	23.61	46.63	93.77	187.84	371.27
	Ratio to BFS	5.18	5.11	3.55	4.81	4.81	4.96	5.04	5.02
2	BFS (sec)	0.31	0.61	1.19	2.34	4.65	9.29	18.66	37.06
	Δ-stepping (sec)	1.92	3.44	6.57	12.72	24.88	48.40	96.15	187.99
	Ratio to BFS	6.25	5.66	5.52	5.43	5.35	5.21	5.15	5.07
	Relative Speedup	1.67	1.84	1.83	1.86	1.87	1.94	1.95	1.99
4	BFS (sec)	0.16	0.31	0.61	1.19	2.37	4.71	9.38	19.59
	Δ-stepping (sec)	1.23	2.07	3.77	7.07	13.63	25.40	50.08	96.89
	Ratio to BFS	7.69	6.68	6.18	5.94	5.75	5.39	5.34	4.95
	Relative Speedup	2.61	3.06	3.20	3.34	3.42	3.69	3.75	3.83
8	BFS (sec)	0.09	0.16	0.31	0.60	1.18	2.35	4.73	9.37
	Δ-stepping (sec)	0.96	1.40	2.39	4.28	8.04	13.81	27.29	49.18
	Ratio to BFS	10.67	8.48	7.74	7.13	6.81	6.88	5.77	5.25
	Relative Speedup	3.34	4.53	5.04	5.52	5.80	6.79	6.88	7.55
16	BFS (sec)	0.06	0.10	0.17	0.32	0.62	1.20	2.39	4.73
	Δ-stepping (sec)	0.84	1.24	1.84	3.06	5.45	8.34	15.91	25.47
	Ratio to BFS	7.55	12.40	10.60	9.22	8.83	6.95	6.66	5.38
	Relative Speedup	3.82	5.11	6.55	7.71	8.55	11.24	11.81	14.58
32	BFS (sec)	0.05	0.07	0.11	0.19	0.36	0.69	1.36	2.68
	Δ-stepping (sec)	0.78	1.047	1.52	2.42	4.12	5.70	10.31	13.90
	Ratio to BFS	15.60	15.00	13.81	12.74	11.44	15.83	7.58	5.19
	Relative Speedup	4.12	6.04	7.93	9.76	11.32	16.45	18.22	26.71
40	BFS (sec)	0.04	0.06	0.10	0.17	0.32	0.61	1.20	2.37
	Δ-stepping (sec)	0.81	1.05	1.53	2.35	3.98	5.15	9.51	11.96
	Ratio to BFS	18.41	16.41	15.30	13.82	12.44	8.44	7.92	5.04
	Relative Speedup	3.96	6.04	7.88	10.05	11.72	11.11	19.75	31.04

Table 16: MTA-2 performance (execution time in seconds, normalized performance with reference to the baseline BFS, relative speedup) of our Δ-stepping implementation for RandomLogUnif4-n graphs. Problem instance denotes the log of the number of vertices. p denotes the number of processors. $m = 4n$ edges and maximum weight $C = n$.

p	Problem Instance	**21**	**22**	**23**	**24**	**25**	**26**	**27**
1	BFS (sec)	0.62	1.24	3.39	4.91	9.70	18.90	37.30
	Δ-stepping (sec)	20.43	41.72	85.10	173.96	378.80	878.86	1687.59
	Ratio to BFS	32.95	33.64	25.10	35.43	39.05	46.50	45.24
4	BFS (sec)	0.16	0.31	0.61	1.19	2.37	4.71	9.38
	Δ-stepping (sec)	6.03	11.17	22.90	45.38	97.63	224.46	426.02
	Ratio to BFS	37.69	36.03	37.54	38.13	41.19	47.65	45.52
	Relative Speedup	3.38	3.73	3.72	3.83	3.88	3.91	3.96
16	BFS (sec)	0.06	0.10	0.17	0.32	0.62	1.20	2.39
	Δ-stepping (sec)	2.47	3.94	7.43	13.96	26.50	60.82	113.12
	Ratio to BFS	41.17	39.40	43.70	43.62	42.74	50.68	47.33
	Relative Speedup	8.27	10.59	11.45	12.46	14.29	14.45	14.92
40	BFS (sec)	0.04	0.06	0.10	0.17	0.32	0.61	1.20
	Δ-stepping (sec)	1.99	2.61	4.27	7.23	12.86	29.58	51.89
	Ratio to BFS	49.17	43.50	42.70	42.53	40.19	48.49	43.24
	Relative Speedup	10.27	15.98	19.93	24.06	29.46	29.71	32.52

Table 17: MTA-2 performance (execution time in seconds, normalized performance with reference to the baseline BFS, relative speedup) of our Δ-stepping implementation on Random4-C graphs. Problem instance denotes the log of the maximum edge weight. p denotes the number of processors. $n = 2^{26}$ vertices, $m = 4n$ edges.

p	Problem Instance	**0**	**3**	**6**	**9**	**12**	**15**
1	BFS (sec)	19.07	19.07	19.07	19.07	19.07	19.07
	Δ-stepping (sec)	93.66	93.74	94.34	93.22	95.76	94.11
	Ratio to BFS	4.91	4.91	4.89	4.89	5.01	4.83
2	BFS (sec)	9.38	9.38	9.38	9.38	9.38	9.38
	Δ-stepping (sec)	48.24	48.15	48.78	48.5	49.25	48.63
	Ratio to BFS	5.14	5.13	5.20	5.17	5.25	5.18
	Relative Speedup	1.94	1.95	1.91	1.92	1.94	1.93
4	BFS (sec)	4.73	4.73	4.73	4.73	4.73	4.73
	Δ-stepping (sec)	25.81	25.43	25.47	25.81	25.39	25.35
	Ratio to BFS	5.46	5.38	5.38	5.46	5.37	5.36
	Relative Speedup	3.63	3.69	3.66	3.61	3.77	3.71
8	BFS (sec)	2.36	2.36	2.36	2.36	2.36	2.36
	Δ-stepping (sec)	14.06	13.67	13.86	13.85	14.07	13.85
	Ratio to BFS	5.96	5.79	5.87	5.87	5.96	5.87
	Relative Speedup	6.66	6.86	6.73	6.73	6.80	6.79
16	BFS (sec)	1.21	1.21	1.21	1.21	1.21	1.21
	Δ-stepping (sec)	8.37	8.38	8.4	8.37	8.42	8.38
	Ratio to BFS	6.92	6.92	6.94	6.92	6.96	6.92
	Relative Speedup	11.19	11.19	11.11	11.14	11.37	11.23
32	BFS (sec)	0.69	0.69	0.69	0.69	0.69	0.69
	Δ-stepping (sec)	5.66	5.65	5.66	5.68	5.66	5.67
	Ratio to BFS	8.20	8.19	8.20	8.23	8.20	8.21
	Relative Speedup	11.42	11.45	11.38	11.32	11.67	11.45
40	BFS (sec)	0.61	0.61	0.61	0.61	0.61	0.61
	Δ-stepping (sec)	5.23	5.27	5.22	5.23	5.21	5.26
	Ratio to BFS	8.52	8.58	8.50	8.52	8.48	8.57
	Relative Speedup	17.91	17.79	17.88	17.82	18.38	17.89

Table 18: MTA-2 performance (execution time in seconds, normalized performance with reference to the baseline BFS) of our Δ-stepping implementation on Long-n graphs. Problem instance denotes the log of the rectangular grid x dimension. p denotes the number of processors, $y = 16$, $n = xy$, $m \approx 4n$ edges, and maximum weight $C = n$.

p	Problem Instance	**10**	**11**	**12**	**13**
1	BFS (sec)	0.54	1.22	1.54	4.19
	Δ-stepping (sec)	3.99	8.57	13.77	32.11
	Ratio to BFS	7.39	7.02	8.94	7.66
4	BFS (sec)	0.74	1.43	2.12	4.52
	Δ-stepping (sec)	5.36	11.20	17.92	42.06
	Ratio to BFS	7.24	7.83	8.45	9.30
16	BFS (sec)	1.04	1.85	3.09	6.72
	Δ-stepping (sec)	7.10	15.07	23.50	56.08
	Ratio to BFS	6.83	8.14	7.60	8.34
40	BFS (sec)	1.31	2.43	4.00	8.29
	Δ-stepping (sec)	12.53	23.64	40.02	90.59
	Ratio to BFS	9.56	9.73	10.00	10.97

p	Problem Instance	**14**	**15**	**16**	**17**
1	BFS (sec)	7.60	14.30	34.90	55.62
	Δ-stepping (sec)	57.16	123.73	243.53	404.91
	Ratio to BFS	7.52	8.65	6.97	7.28
4	BFS (sec)	9.27	19.80	39.48	71.49
	Δ-stepping (sec)	73.93	158.72	306.69	567.63
	Ratio to BFS	7.97	8.01	7.77	7.94
16	BFS (sec)	13.56	25.44	57.71	107.00
	Δ-stepping (sec)	97.99	212.51	503.33	967.70
	Ratio to BFS	7.23	8.35	8.72	9.04
40	BFS (sec)	18.14	32.33	72.99	132.36
	Δ-stepping (sec)	171.13	330.72	812.02	1534.05
	Ratio to BFS	9.43	10.23	11.12	11.59

Table 19: MTA-2 performance (execution time in seconds, normalized performance with reference to the baseline BFS) of our Δ-stepping implementation on Long-C graphs. Problem instance denotes the log of the maximum edge weight. p denotes the number of processors. The grid dimensions are given by $x = 2^{14}$ and $y = 16$.

p	Problem Instance	**0**	**3**	**6**	**9**	**12**	**15**
1	BFS (sec)	7.60	7.60	7.60	7.60	7.60	7.60
	Δ-stepping (sec)	57.24	56.88	57.13	57.89	58.11	56.97
	Ratio to BFS	7.53	7.48	7.52	7.62	7.65	7.50
4	BFS (sec)	9.27	9.27	9.27	9.27	9.27	9.27
	Δ-stepping (sec)	74.02	73.88	73.92	74.68	75.17	75.49
	Ratio to BFS	7.98	7.97	7.97	8.06	8.10	8.14
16	BFS (sec)	13.56	13.56	13.56	13.56	13.56	13.56
	Δ-stepping (sec)	96.76	97.11	97.45	98.82	98.30	98.61
	Ratio to BFS	7.14	7.16	7.19	7.24	7.25	7.27
40	BFS (sec)	18.14	18.14	18.14	18.14	18.14	18.14
	Δ-stepping (sec)	172.00	171.34	173.43	172.84	172.49	173.19
	Ratio to BFS	9.48	9.44	9.56	9.53	9.51	9.55

Table 20: MTA-2 performance (execution time in seconds, normalized performance with reference to the baseline BFS) of our Δ-stepping implementation on Square-n graphs. Problem instance denotes the log of the number of the grid dimension x. p denotes the number of processors. $x = y$, $n = xy$, $m \approx 4n$ edges, and maximum weight $C = n$.

p	Problem Instance	**6**	**7**	**8**	**9**	**10**	**11**	**12**
1	BFS (sec)	0.05	0.12	0.24	0.57	1.32	3.22	8.55
	Δ-stepping (sec)	0.20	0.52	1.28	2.80	7.84	20.56	68.33
	Ratio to BFS	4.00	4.33	5.33	4.91	5.94	6.38	7.99
4	BFS (sec)	0.09	0.16	0.33	0.72	1.51	3.19	6.59
	Δ-stepping (sec)	0.23	0.60	1.41	2.84	6.85	14.29	38.62
	Ratio to BFS	2.55	3.75	4.27	3.94	4.54	4.48	5.86
16	BFS (sec)	0.11	0.22	0.41	0.95	1.99	3.93	7.68
	Δ-stepping (sec)	0.28	0.73	1.64	3.3	7.83	14.93	35.51
	Ratio to BFS	2.54	3.32	4.00	3.47	3.93	3.80	4.62
40	BFS (sec)	0.12	0.23	0.44	1.00	2.05	4.01	7.90
	Δ-stepping (sec)	0.35	0.84	1.91	3.59	8.35	15.29	35.46
	Ratio to BFS	2.92	3.65	4.34	3.59	4.07	3.81	4.49

Table 21: MTA-2 performance (execution time in seconds) of the baseline BFS, and our Δ-stepping implementation on the USA core road networks with distance (road-d) and transit times (road-t) as the length function.

p	Instance	**CTR**	**W**	**E**	**LKS**	**CAL**	**NE**
1	BFS time (sec)	7.69	5.19	3.95	3.38	2.39	2.01
	road-d time (sec)	49.89	32.91	23.46	15.08	13.09	14.33
	road-t time (sec)	37.06	24.01	15.12	14.51	11.32	6.66
4	BFS time (sec)	6.48	4.95	4.09	3.6	2.53	2.14
	road-d time (sec)	48.58	30.12	23.29	15.59	13.92	14.21
	road-t time (sec)	34.38	23.75	15.73	15.36	12.03	7.18
16	BFS time (sec)	7.26	5.85	4.94	4.32	3.02	2.53
	road-d time (sec)	52.83	36.74	26.91	18.94	15.96	15.03
	road-t time (sec)	39.95	27.86	18.53	18.21	14.25	8.54
40	BFS time (sec)	7.50	6.56	5.47	4.43	3.37	2.92
	road-d time (sec)	55.19	39.84	33.32	21.66	18.15	16.23
	road-t time (sec)	42.94	30.24	21.15	20.09	16.29	9.96

p	Instance	**NW**	**FLA**	**COL**	**BAY**	**NY**
1	BFS time (sec)	1.52	1.98	1.51	0.72	0.67
	road-d time (sec)	13.80	12.76	6.52	3.16	2.70
	road-t time (sec)	7.06	9.22	5.03	2.34	1.69
4	BFS time (sec)	1.57	2.15	1.76	0.83	0.76
	road-d time (sec)	13.41	12.28	7.90	3.70	3.39
	road-t time (sec)	8.07	10.45	5.95	2.73	1.91
16	BFS time (sec)	1.86	2.56	2.13	1.01	0.94
	road-d time (sec)	14.06	15.28	8.81	4.62	4.08
	road-t time (sec)	9.64	12.41	7.14	3.25	2.31
40	BFS time (sec)	2.18	2.95	2.19	1.05	0.95
	road-d time (sec)	15.11	16.44	9.93	5.31	5.06
	road-t time (sec)	12.05	14.54	9.09	3.95	2.82

Table 22: MTA-2 performance (execution time in seconds) of our Δ-stepping implementation on the full USA and Europe road networks.

p	Instance	**USA-d**	**USA-t**	**Europe-d**	**Europe-t**
1	BFS (sec)	10.04	9.93	7.04	7.05
	Δ-stepping (sec)	182.84	142.19	177.84	79.40
4	BFS (sec)	8.23	7.96	5.18	5.20
	Δ-stepping (sec)	164.29	127.81	135.49	63.04
16	BFS (sec)	10.21	10.14	5.89	6.01
	Δ-stepping (sec)	167.83	137.52	137.67	64.36
40	BFS (sec)	10.39	10.69	6.06	5.94
	Δ-stepping (sec)	173.11	150.63	143.13	67.16

REFERENCES

[1] Abou-Rjeili, A. and Karypis, G., "Multilevel algorithms for partitioning power-law graphs," in *Proc. 20th Int'l Parallel and Distributed Proc. Symp. (IPDPS 2006)*, (Rhodes, Greece), Apr. 2006.

[2] Adamson, P. and Tick, E., "Greedy partitioned algorithms for the shortest path problem," *Int'l Journal of Parallel Programming*, vol. 20, pp. 271–298, August 1991.

[3] Ajwani, D., Meyer, U., and Osipov, V., "Improved external memory BFS implementations," in *Proc. The 9th Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*, (New Orleans, LA), pp. 3–12, Jan. 2007.

[4] Albert, R. and Barabási, A.-L., "Statistical mechanics of complex networks," *Reviews of Modern Physics*, vol. 74, no. 1, pp. 47–97, 2002.

[5] Alfarano, C. *et al.*, "The biomolecular interaction network database and related tools: 2005 update," *Nucleic Acids Research*, vol. 33, pp. D418–D424, 2005.

[6] Amaral, L., Scala, A., Barthélémy, M., and Stanley, H., "Classes of small-world networks," *Proc. National Academy of Sciences USA*, vol. 97, no. 21, pp. 11149–11152, 2000.

[7] AMD, "Multi-Core Processors from AMD." http://multicore.amd.com/, 2008.

[8] Analytic Technologies, "UCINET 6 social network analysis software." http://www.analytictech.com/ucinet/ucinet.htm, 2008.

[9] Asanovic, K. *et al.*, "The landscape of parallel computing research: A view from Berkeley," Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 2006.

[10] Bader, D., Heitsch, C., and Madduri, K., "Betweenness centrality on an integer torus." submitted, 2008.

[11] Bader, D., Illendula, A., Moret, B. M., and Weisse-Bernstein, N., "Using PRAM algorithms on a uniform-memory-access shared-memory architecture," in *Proc. 5th Int'l Workshop on Algorithm Engineering (WAE 2001)* (Brodal, G., Frigioni, D., and Marchetti-Spaccamela, A., eds.), vol. 2141 of *Lecture Notes in Computer Science*, (Århus, Denmark), pp. 129–144, Springer-Verlag, Aug. 2001.

[12] Bader, D., Kintali, S., Madduri, K., and Mihail, M., "Approximating betweenness centrality," in *Proc. 5th Workshop on Algorithms and Models for the Web-Graph (WAW 2007)* (Bonato, A. and Chung, F., eds.), vol. 4863 of *Lecture Notes in Computer Science*, (San Diego, CA), pp. 124–137, Springer-Verlag, Dec. 2007.

[13] Bader, D. and Madduri, K., "Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors," in *Proc. 12th Int'l Conf. on High Performance Computing (HiPC 2005)*, (Goa, India), Springer-Verlag, Dec. 2005.

[14] Bader, D. and Madduri, K., "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 2006)*, (Columbus, OH), pp. 523–530, IEEE Computer Society, Aug. 2006.

[15] Bader, D. and Madduri, K., "Parallel algorithms for evaluating centrality indices in real-world networks," in *Proc. 35th Int'l Conf. on Parallel Processing (ICPP 2006)*, (Columbus, OH), pp. 539–550, IEEE Computer Society, Aug. 2006.

[16] Bader, D. and Madduri, K., "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.

[17] Bader, D. and Madduri, K., "SNAP, Small-world Network Analysis and Partitioning: an open-source parallel graph framework for the exploration of large-scale networks," in *Proc. 22nd Int'l Parallel and Distributed Processing Symp. (IPDPS 2008)*, (Miami, FL), Apr. 2008.

[18] Bader, D., Madduri, K., Gilbert, J., Shah, V., Kepner, J., Meuse, T., and Krishnamurthy, A., "Designing scalable synthetic compact applications for benchmarking high productivity computing systems," *CTWatch Quarterly*, vol. 2, Nov. 2006.

[19] Bader, D., Sreshta, S., and Weisse-Bernstein, N., "Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs)," in *Proc. 9th Int'l Conf. on High Performance Computing (HiPC 2002)* (Sahni, S., Prasanna, V., and Shukla, U., eds.), vol. 2552 of *Lecture Notes in Computer Science*, (Bangalore, India), pp. 63–75, Springer-Verlag, Dec. 2002.

[20] Barabási, A.-L., "Network database." http://www.barabasilab.com/resources.php, 2008.

[21] Barabási, A.-L. and Albert, R., "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[22] Barnes, G. and Ruzzo, W., "Deterministic algorithms for undirected s-t connectivity using polynomial time and sublinear space," in *Proc. 23rd Ann. ACM Symp. on Theory of Computing (STOC)*, (New Orleans, LA), pp. 43–53, May 1991.

[23] Barroso, L., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., and Verghese, B., "Piranha: A scalable architecture based on single-chip multi-processing," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 282–293, 2000.

[24] Batada, N., Reguly, T., Breitkreutz, A., Boucher, L., Breitkreutz, B.-J., Hurst, L., and Tyers, M., "Stratus Not Altocumulus: A New View of the Yeast Protein Interaction Network," *PLoS Biology*, vol. 4, p. e317, Oct. 2006.

[25] Batagelj, V. and Mrvar, A., "Pajek – program for large network analysis," *Connections*, vol. 21, no. 2, pp. 47–57, 1998.

[26] Batagelj, V. and Mrvar, A., "PAJEK datasets." http://www.vlado.fmf.uni-lj.si/pub/networks/data/, 2006.

[27] Bavelas, A., "Communication patterns in task oriented groups," *J. Acoustical Soc. of America*, vol. 22, pp. 271–282, 1950.

[28] Berg, J., Lassig, M., and Wagner, A., "Structure and evolution of protein interaction networks: a statistical model for link dynamics and gene duplications," *BMC Evolutionary Biology*, vol. 4, no. 1, p. 51, 2004.

[29] Boldi, P. and Vigna, S., "The webgraph framework I: compression techniques," in *Proc. 13th Intl. Conf. on World Wide Web (WWW 13)*, (New York, NY, USA), pp. 595–602, ACM Press, 2004.

[30] Bork, P., Jensen, L., von Mering, C., A.K.Ramani, Lee, I., and Marcotte, E., "Protein interaction networks from yeast to human," *Curr. Opin. Struct. Bio.*, vol. 14, pp. 292–299, 2004.

[31] Brandes, U., "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[32] Brandes, U., Delling, D., Gaertler, M., Görke, R., Höfer, M., Nikoloski, Z., and Wagner, D., "On finding graph clusterings with maximum modularity," in *Proc. 33rd Intl. Workshop on Graph-Theoretic Concepts in CS (WG 2007)*, (Dornburg, Germany), June 2007.

[33] Brandes, U., Gaertler, M., and Wagner, D., "Engineering graph clustering: Models and experimental evaluation," *Journal of Experimental Algorithmics*, vol. 12, p. 1.1, 2007.

[34] Brandes, U. and Pich, C., "Centrality estimation in large networks," *Intl. Journal of Bifurcation and Chaos, Special Issue on Complex Networks' Structure and Dynamics*, pp. 2303–1318, 2007.

[35] Briggs, P. and Torczon, L., "An efficient representation for sparse sets," *ACM Letters on Programming Languages and Systems*, vol. 2, no. 1-4, pp. 59–69, 1993.

[36] Brin, S. and Page, L., "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1–7, pp. 107–117, 1998.

[37] Brodal, G., Träff, J., and Zaroliagis, C., "A parallel priority queue with constant time operations," *Journal of Parallel and Distributed Computing*, vol. 49, no. 1, pp. 4–21, 1998.

[38] Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., and Wiener, J., "Graph structure in the web," *Computer Networks*, vol. 33, no. 1-6, pp. 309–320, 2000.

[39] Buckley, N. and van Alstyne, M., "Does email make white collar workers more productive?," tech. rep., University of Michigan, 2004.

[40] Callaway, D., Newman, M., Strogatz, S., and Watts, D., "Network robustness and fragility: percolation on random graphs," *Physical Review Letters*, vol. 85, no. 25, pp. 5468–5471, 2000.

[41] Chakrabarti, D., Zhan, Y., and Faloutsos, C., "R-MAT: A recursive model for graph mining," in *Proc. 4th SIAM Intl. Conf. on Data Mining (SDM 2004)*, (Orlando, FL), pp. 1–5, SIAM, Apr. 2004.

[42] Chandy, K. and Misra, J., "Distributed computation on graphs: Shortest path algorithms," *Communications of the ACM*, vol. 25, no. 11, pp. 833–837, 1982.

[43] Cherkassky, B., Goldberg, A., and Radzik, T., "Shortest paths algorithms: theory and experimental evaluation," *Mathematical Programming*, vol. 73, pp. 129–174, 1996.

[44] Cisic, D., Kesic, B., and Jakomin, L., "Research of the power in the supply chain," International Trade, Economics Working Paper Archive EconWPA, Apr. 2000.

[45] Clauset, A., Newman, M., and Moore, C., "Finding community structure in very large networks," *Physical Review E*, vol. 70, p. 066111, 2004.

[46] Coffman, T., Greenblatt, S., and Marcus, S., "Graph-based technologies for intelligence analysis," *Communications of the ACM*, vol. 47, no. 3, pp. 45–47, 2004.

[47] Cohen, E., "Using selective path-doubling for parallel shortest-path computation," *Journal of Algorithms*, vol. 22, no. 1, pp. 30–56, 1997.

[48] Cohen, R., Erez, K., ben-Avraham, D., and Havlin, S., "Breakdown of the internet under intentional attack," *Physical Review Letters*, vol. 86, no. 16, pp. 3682–3685, 2001.

[49] Cormen, T. H., Leiserson, C. E., and Rivest, R. L., *Introduction to Algorithms*. Cambridge, MA: MIT Press, Inc., 1990.

[50] Cray, Inc., "Cray XMT platform." http://www.cray.com/products/xmt/, 2007.

[51] Crobak, J., Berry, J., Madduri, K., and Bader, D., "Advanced shortest path algorithms on a massively-multithreaded architecture," in *Proc. Workshop on Multithreaded Architectures and Applications (MTAAP 2007)*, (Long Beach, CA), pp. 1–8, IEEE Computer Society, March 2007.

[52] Dasgupta, A., Hopcroft, J., and McSherry, F., "Spectral analysis of random graphs with skewed degree distributions," in *Proc. 45th Ann. IEEE Symp. on Foundations of Computer Science (FOCS 2004)*, (Washington, DC), pp. 602–610, IEEE Computer Society, 2004.

[53] Davis, T., "University of Florida Sparse Matrix Collection." http://www.cise.ufl.edu/research/sparse/matrices, 2007.

[54] del Sol, A., Fujihashi, H., and O'Meara, P., "Topology of small-world networks of protein-protein complex structures," *Bioinformatics*, vol. 21, no. 8, pp. 1311–1315, 2005.

[55] Demetrescu, C., Finocchi, I., and Italiano, G., "Dynamic graphs," in *Handbook on Data Structures and Applications* (Mehta, D. and Sahni, S., eds.), ch. 36, CRC Press, 2005.

[56] Demetrescu, C., Finocchi, I., and Ribichini, A., "Trading off space for passes in graph streaming problems," in *Proc. 17th Ann. Symp. on Discrete Algorithms (SODA 2006)*, (Miami, FL), pp. 714–723, ACM Press, Jan. 2006.

[57] Demetrescu, C., Goldberg, A., and Johnson, D., "9th DIMACS implementation challenge – Shortest Paths." http://www.dis.uniroma1.it/~challenge9/, 2006.

[58] Demetrescu, C., Goldberg, A., and Johnson, D., "9th DIMACS implementation challenge – Shortest Paths: Reference benchmark package." http://www.dis.uniroma1.it/~challenge9/download.shtml, 2006.

[59] D.Eppstein, Galil, Z., and Italiano, G., "Dynamic graph algorithms," in *Handbook of Algorithms and Theory of Computation* (Atallah, M., ed.), ch. 8, CRC Press, November 1998.

[60] Dial, R., "Algorithm 360: Shortest path forest with topological ordering," *Communications of the ACM*, vol. 12, pp. 632–633, 1969.

[61] Dial, R., Glover, F., Karney, D., and Klingman, D., "A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees," *Networks*, vol. 9, pp. 215–248, 1979.

[62] Dijkstra, E., "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[63] Doreian, P. and Albert, L., "Partitioning political actor networks: Some quantitative tools for analyzing qualitative networks," *J. Quantitative Anthropology*, vol. 1, pp. 279–291, 1989.

[64] Driscoll, J., Gabow, H., Shrairman, R., and Tarjan, R., "Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation," *Communications of the ACM*, vol. 31, no. 11, pp. 1343–1354, 1988.

[65] Duch, J. and Arenas, A., "Community detection in complex networks using extremal optimization," *Physical Review E*, vol. 72, p. 027104, 2005.

[66] Eppstein, D., Galil, Z., Italiano, G., and Nissenzweig, A., "Sparsification: a technique for speeding up dynamic graph algorithms," *Journal of the ACM*, vol. 44, no. 5, pp. 669–696, 1997.

[67] Eppstein, D. and Wang, J., "Fast approximation of centrality," in *Proc. 12th Ann. Symp. on Discrete Algorithms (SODA 2001)*, (Washington, DC), pp. 228–229, 2001.

[68] Erdős, P. and Rényi, A., "On random graphs I," *Publicationes Mathematicae*, vol. 6, p. 1959, 290–297.

[69] Faloutsos, M., Faloutsos, P., and Faloutsos, C., "On power-law relationships of the Internet topology," in *Proc. ACM SIGCOMM 1999*, (Cambridge, MA), pp. 251–262, ACM, Aug. 1999.

[70] Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., and Zhang, J., "On graph problems in a semi-streaming model," *Theoretical Computer Science*, vol. 348, no. 2, pp. 207–216, 2005.

[71] Feo, J., Harper, D., Kahan, S., and Konecny, P., "ELDORADO," in *Proc. 2nd Conf. on Computing Frontiers* (Bagherzadeh, N., Valero, M., and Ramírez, A., eds.), (Ischia, Italy), pp. 28–34, ACM, 2005.

[72] Fredman, M. and Tarjan, R., "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, pp. 596–615, 1987.

[73] Fredman, M. and Willard, D., "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," *Journal of Computer and System Sciences*, vol. 48, pp. 533–551, 1994.

[74] Freeman, L., "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, pp. 35–41, 1977.

[75] Freeman, L., *The development of social network analysis: a study in the sociology of science*. BookSurge Publishers, 2004.

[76] Frieze, A. and Rudolph, L., "A parallel algorithm for all-pairs shortest paths in a random graph," in *Proc. 22nd Allerton Conf. on Communication, Control and Computing*, (Monticello, IL), pp. 663–670, 1984.

[77] Gallo, G. and Pallottino, P., "Shortest path algorithms," *Annals of Operations Research*, vol. 13, pp. 3–79, 1988.

[78] Gandhi, T. *et al.*, "Analysis of the human protein interactome and comparison with yeast, worm and fly interaction datasets," *Nature Genetics*, vol. 38, pp. 285–293, 2006.

[79] Garey, M. and Johnson, D., *Computers and Intractability – A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.

[80] Gazit, H. and Miller, G. L., "An improved parallel algorithm that computes the BFS numbering of a directed graph," *Information Processing Letters*, vol. 28, no. 2, pp. 61–65, 1988.

[81] Giot, L. *et al.*, "A protein interaction map of drosophila melanogaster," *Science*, vol. 302, pp. 1727–1736, 2003.

[82] Girvan, M. and Newman, M., "Community structure in social and biological networks," *Proc. National Academy of Sciences USA*, vol. 99, no. 12, pp. 7821–7826, 2002.

[83] Glover, F., Glover, R., and Klingman, D., "Computational study of an improved shortest path algorithm," *Networks*, vol. 14, pp. 23–37, 1984.

[84] Goldberg, A., "Shortest path algorithms: Engineering aspects," in *Proc. 12th Int'l Symp. on Algorithms and Computation (ISAAC 2001)*, (London, UK), pp. 502–513, Springer-Verlag, 2001.

[85] Goldberg, A., "A simple shortest path algorithm with linear average time," in *9th Annual European Symp. on Algorithms (ESA 2001)*, vol. 2161 of *Lecture Notes in Computer Science*, (Aachen, Germany), pp. 230–241, Springer, 2001.

[86] Graham, R., Knuth, D., and Patashnik, O., *Concrete mathematics*. Reading, MA: Addison-Wesley Publishing Company Advanced Book Program, 1989. A foundation for computer science.

[87] Gregor, D. and Lumsdaine, A., "Lifting sequential graph algorithms for distributed-memory parallel computation," in *Proc. 20th ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*, (New York, NY), pp. 423–437, ACM Press, 2005.

[88] Guillaume, J.-L. and Latapy, M., "Bipartite graphs as models of complex networks," in *Proc. 1st Int'l Workshop on Combinatorial and Algorithmic Aspects of Networking*, vol. 3405 of *Lecture Notes in Computer Science*, (Alberta, Canada), pp. 127–139, Springer-Verlag, Aug. 2004.

[89] Guimerà, R., Mossa, S., Turtschi, A., and Amaral, L., "The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles," *Proc. National Academy of Sciences USA*, vol. 102, no. 22, pp. 7794–7799, 2005.

[90] Hagerup, T., "Improved shortest paths on the word RAM," in *27th Colloquium on Automata, Languages and Programming (ICALP 2000)*, vol. 1853 of *Lecture Notes in Computer Science*, (Geneva, Switzerland), pp. 61–72, Springer-Verlag, 2000.

[91] Han, Y., Pan, V., and Reif, J., "Efficient parallel algorithms for computing the all pair shortest paths in directed graphs," *Algorithmica*, vol. 17, no. 4, pp. 399–415, 1997.

[92] Held, J., Bautista, J., and Koehl, S., "From a few cores to many: A tera-scale computing research overview." http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf, 2008.

[93] Helman, D. R. and JáJá, J., "Designing practical efficient algorithms for symmetric multiprocessors," in *1st Workshop on Algorithm Engineering and Experimentation (ALENEX 1999)*, vol. 1619 of *Lecture Notes in Computer Science*, (Baltimore, MD), pp. 37–56, Springer-Verlag, Jan. 1999.

[94] Helman, D. R. and JáJá, J., "Prefix computations on symmetric multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 61, no. 2, pp. 265–278, 2001.

[95] Hendrickson, B. and Leland, R., "A multilevel algorithm for partitioning graphs," in *Proc. Supercomputing '95*, (San Diego, CA), Dec. 1995.

[96] Henzinger, M. and King, V., "Randomized dynamic graph algorithms with polylogarithmic time per operation," in *Proc. 27th Ann. Symp. of Theory of Computing (STOC 1995)*, (New York, NY, USA), pp. 519–527, ACM Press, 1995.

[97] Henzinger, M., Raghavan, P., and Rajagopalan, S., "Computing on data streams," Tech. Rep. TR-1998-011, Compaq Systems Research Center, Palo Alto, CA, May 1998.

[98] Hermjakob, H., Montecchi-Palazzi, L., Lewington, C., Mudali, S., Kerrien, S., Orchard, S., Vingron, M., Roechert, B., Roepstorff, P., Valencia, A., Margalit, H., Armstrong, J., Bairoch, A., Cesareni, G., Sherman, D., and Apweiler, R., "IntAct: an open source molecular interaction database," *Nucleic Acids Res.*, vol. 32, pp. D452–D455, 2004.

[99] Hoeffding, W., "Probability inequalities for sums of bounded random variables," *J. American Statistical Association*, vol. 58, no. 301, pp. 13–30, 1963.

[100] Holme, P., "Congestion and centrality in traffix flow on complex networks," *Advances in Complex Systems*, vol. 6, no. 2, pp. 163–176, 2003.

[101] Hribar, M. and Taylor, V., "Performance study of parallel shortest path algorithms: Characteristics of good decomposition," in *Proc. 13th Annual Conf. of Intel Supercomputers Users Group*, (Albuquerque, NM), pp. 1–27, 1997.

[102] Hribar, M., Taylor, V., and Boyce, D., "Parallel shortest path algorithms: Identifying the factors that affect performance," Report CPDC-TR-9803-015, Northwestern University, Evanston, IL, 1998.

[103] Hribar, M., Taylor, V., and Boyce, D., "Reducing the idle time of parallel shortest path algorithms," Report CPDC-TR-9803-016, Northwestern University, Evanston, IL, 1998.

[104] Hribar, M., Taylor, V., and Boyce, D., "Termination detection for parallel shortest path algorithms," *Journal of Parallel and Distributed Computing*, vol. 55, pp. 153–165, 1998.

[105] Inokuchi, A., Washio, T., and Motoda, H., "An apriori-based algorithm for mining frequent substructures from graph data," in *Proc. 4th European Conf. on Principles of Data Mining and Knowledge Discovery (PKDD)*, (Lyon, France), pp. 13–23, Sept. 2000.

[106] Intel, "Intel Multi-Core Technology." http://www.intel.com/multi-core/, 2008.

[107] JáJá, J., *An Introduction to Parallel Algorithms*. New York: Addison-Wesley Publishing Company, 1992.

[108] Jeong, H., Mason, S., Barabási, A.-L., and Oltvai, Z., "Lethality and centrality in protein networks," *Nature*, vol. 411, pp. 41–42, 2001.

[109] Jeong, H., Oltvai, Z., and Barabási, A.-L., "Prediction of protein essentiality based on genomic data," *ComPlexUs*, vol. 1, pp. 19–28, 2003.

[110] Joy, M., Brock, A., Ingber, D., and Huang, S., "High-betweenness proteins in the yeast protein interaction network," *Journal of Biomedicine and Biotechnology*, vol. 2, pp. 96–103, 2005.

[111] Kahle, J., Day, M., Hofstee, H., Johns, C., Maeurer, T., and Shippy, D., "Introduction to the Cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, 2005.

[112] Kannan, R., Vempala, S., and Vetta, A., "On clusterings: Good, bad and spectral," *Journal of the ACM*, vol. 51, no. 3, pp. 497–515, 2004.

[113] Karypis, G., Han, E., and Kumar, V., "Chameleon: Hierarchical clustering using dynamic modeling," *IEEE Computer*, vol. 32, no. 8, pp. 68–75, 1999.

[114] Karypis, G. and Kumar, V., *MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*. Department of Computer Science, University of Minnesota, version 4.0 ed., Sept. 1998.

[115] Karypis, G. and Kumar, V., "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 96–129, 1998.

[116] Kempe, D., Kleinberg, J., and Kumar, A., "Connectivity and inference problems for temporal networks," *Journal of Computer and System Sciences*, vol. 64, no. 4, pp. 820–842, 2002.

[117] Kernighan, B. and Lin, S., "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[118] Klein, P. and Subramanian, S., "A randomized parallel algorithm for single-source shortest paths," *Journal of Algorithms*, vol. 25, no. 2, pp. 205–220, 1997.

[119] Kongetira, P., Aingaran, K., and Olukotun, K., "Niagara: A 32-way multithreaded Sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[120] Krebs, V., "InFlow 3.1 - social network mapping software." http://www.orgnet.com, 2005.

[121] Krebs, V., "Mapping networks of terrorist cells," *Connections*, vol. 24, no. 3, pp. 43–52, 2002.

[122] Ladner, R., Fix, J., and LaMarca, A., "The cache performance of traversals and random accesses," in *Proc. 10th Ann. Symp. Discrete Algorithms (SODA-99)*, (Baltimore, MD), pp. 613–622, ACM-SIAM, 1999.

[123] Ladner, R. E., Fortna, R., and Nguyen, B.-H., "A comparison of cache aware and cache oblivious static search trees using program instrumentation," in *Experimental Algorithmics* (Fleischer, R., Meineche-Schmidt, E., and Moret, B., eds.), vol. 2547 of *Lecture Notes in Computer Science*, pp. 78–92, Springer-Verlag, 2002.

[124] Lang, K., "Finding good nearly balanced cuts in power law graphs," tech. rep., Yahoo! Research, 2004.

[125] Lang, K., "Fixing two weaknesses of the spectral method," in *Proc. Advances in Neurals Information Proc. Systems 18 (NIPS 2005)*, (Vancouver, Canada), Dec. 2005.

[126] Lehner, B. and Fraser, A., "A first-draft human protein-interaction map," *Genome Biology*, vol. 5, no. 9, p. R63, 2004.

[127] Lengauer, T., *Combinatorial algorithms for integrated circuit layout*. John Wiley & Sons, Inc., 1990.

[128] Li, S. *et al.*, "A map of the interactome network of the metazoan *C. elegans*," *Science*, vol. 303, no. 5657, pp. 540–543, 2004.

[129] Liljeros, F., Edling, C., Amaral, L., Stanley, H., and Åberg, Y., "The web of human sexual contacts," *Nature*, vol. 411, pp. 907–908, 2001.

[130] Lipton, R. and Naughton, J., "Estimating the size of generalized transitive closures.," in *Proc. The 15th Int'l. Conf. on Very Large Data Bases (VLDB 1989)*, (Amsterdam, The Netherlands), pp. 165–171, Aug. 1989.

[131] Madduri, K., "9th DIMACS implementation challenge: Shortest Paths. Δ-stepping C/MTA-2 code." http://www.cc.gatech.edu/~kamesh/research/DIMACS-ch9, 2008.

[132] Madduri, K., "SNAP: Small-world network analysis and partitioning." http://snap-graph.sourceforge.net, 2008.

[133] Madduri, K. and Bader, D., "GTgraph: A suite of synthetic graph generators." http://www.cc.gatech.edu/~kamesh/GTgraph, 2008.

[134] Madduri, K., Bader, D., Berry, J., and Crobak, J., "An experimental study of a parallel shortest path algorithm for solving large-scale graph instances," in *Proc. The 9th Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*, (New Orleans, LA), pp. 23–35, SIAM, Jan. 2007.

[135] Meyer, U., "Heaps are better than buckets: parallel shortest paths on unbalanced graphs," in *Proc. 7th Intl. Euro-Par Conference (Euro-Par 2001)*, (Manchester, United Kingdom), pp. 343–351, Springer-Verlag, 2000.

[136] Meyer, U., "Buckets strike back: Improved parallel shortest-paths," in *Proc. 16th Int'l Parallel and Distributed Processing Symp. (IPDPS 2002)*, (Fort Lauderdale, FL), pp. 1–8, IEEE Computer Society, Apr. 2002.

[137] Meyer, U., *Design and Analysis of Sequential and Parallel Single-Source Shortest-Paths Algorithms*. PhD thesis, Universität Saarlandes, Saarbrücken, Germany, Oct. 2002.

[138] Meyer, U., "Average-case complexity of single-source shortest-paths algorithms: lower and upper bounds," *Journal of Algorithms*, vol. 48, no. 1, pp. 91–134, 2003.

[139] Meyer, U. and Sanders, P., "Parallel shortest path for arbitrary graphs," in *Proc. 6th Intl. Euro-Par Conference (Euro-Par 2000)*, vol. 1900 of *Lecture Notes in Computer Science*, (Munich, Germany), pp. 461–470, Springer-Verlag, 2000.

[140] Meyer, U. and Sanders, P., "Δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.

[141] Mihail, M. and Papadimitriou, C., "On the eigenvalue power law," in *Proc. 6th Intl. Workshop on Randomization and Approximation Techniques (RANDOM 2002)* (Rolim, J. and Vadhan, S., eds.), Springer-Verlag, September 2002.

[142] Moret, B., Bader, D., and Warnow, T., "High-performance algorithm engineering for computational phylogenetics," in *Proc. Int'l Conf. on Computational Science*, vol. 2073–2074 of *Lecture Notes in Computer Science*, (San Francisco, CA), Springer-Verlag, 2001.

[143] Muthukrishnan, S., "Data streams: algorithms and applications," *Foundations and Trends in Theoretical Computer Science*, vol. 1, pp. 117–236, August 2005.

[144] Newman, M., "Scientific collaboration networks: II. shortest paths, weighted networks and centrality," *Physical Review E*, vol. 64, p. 016132, 2001.

[145] Newman, M., "The structure and function of complex networks," *SIAM Review*, vol. 45, no. 2, pp. 167–256, 2003.

[146] Newman, M., "Modularity and community structure in networks," *Proc. Nat. Academy of Sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.

[147] Newman, M. and Girvan, M., "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, p. 026113, 2004.

[148] Newman, M., Strogatz, S., and Watts, D., "Random graph models of social networks," *Proc. National Academy of Sciences USA*, vol. 99, no. (suppl. 1), pp. 2566–2572, 2002.

[149] Nieminen, U., "On the centrality in a directed graph," *Social Science Research*, vol. 2, pp. 371–378, 1973.

[150] Palmer, C. and Steffan, J., "Generating network topologies that obey power laws," in *Proc. IEEE Global Internet Symp. (GLOBECOM)*, (San Francisco, CA), pp. 434–438, Nov. 2000.

[151] Papaefthymiou, M. and Rodrigue, J., "Implementing parallel shortest-paths algorithms," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 30, pp. 59–68, 1997.

[152] Park, J., Penner, M., and Prasanna, V., "Optimizing graph algorithms for improved cache performance," in *Proc. 16th Int'l Parallel and Distributed Processing Symp. (IPDPS 2002)*, (Fort Lauderdale, FL), IEEE Computer Society, Apr. 2002.

[153] Pastor-Satorras, R. and Vespignani, A., "Epidemic spreading in scale-free networks," *Physical Review Letters*, vol. 86, no. 14, pp. 3200–3203, 2001.

[154] Peri, S. *et al.*, "Development of human protein reference database as an initial platform for approaching systems biology in humans," *Genome Research*, vol. 13, pp. 2363–2371, 2003.

[155] Pinney, J., McConkey, G., and Westhead, D., "Decomposition of biological networks using betweenness centrality," in *Proc. 9th Annual Int'l Conf. on Research in Computational Molecular Biology (RECOMB 2005)*, (Cambridge, MA), May 2005. Poster session.

[156] PTV Europe, "European road graphs." http://i11www.iti.uni-karlsruhe.de/resources/roadgraphs/index.php, 2006.

[157] Ramalingam, G. and Reps, T., "On the computational complexity of dynamic graph problems," *Theoretical Computer Science*, vol. 158, no. 1-2, pp. 233–277, 1996.

[158] Raman, R., "Recent results on single-source shortest paths problem," *SIGACT News*, vol. 28, pp. 61–72, 1997.

[159] Ramani, A., Bunescu, R., Mooney, R., and Marcotte, E., "Consolidating the set of known human protein-protein interactions in preparation for large-scale mapping of the human interactome," *Genome Biology*, vol. 6, no. 5, p. R40, 2005.

[160] Reguly, T., Breitkreutz, A., Boucher, L., Breitkreutz, B.-J., Hon, G. C., Myers, C. L., Parsons, A., Friesen, H., Oughtred, R., Tong, A., Stark, C., Ho, Y., Botstein, D., Andrews, B., Boone, C., Troyanskya, O., Ideker, T., Dolinski, K., Batada, N., and Tyers, M., "Comprehensive curation and analysis of global interaction networks in Saccharomyces Cerevisia," *Journal of Biology*, vol. 5, p. 11, 2006.

[161] Richards, W., "International network for social network analysis." http://www.insna.org, 2005.

[162] Rual, J.-F. *et al.*, "Towards a proteome-scale map of the human protein-protein interaction network," *Nature*, vol. 437, pp. 1173–1178, 2005.

[163] Sabidussi, G., "The centrality index of a graph," *Psychometrika*, vol. 31, pp. 581–603, 1966.

[164] Salwinski, L., Miller, C., Smith, A., Pettit, F., Bowie, J., and Eisenberg, D., "DIP: the database of interacting proteins: 2004 update," *Nucleic Acids Research*, vol. 32, pp. D449–451, 2004.

[165] Scott, J., *Social Network Analysis: A Handbook*. Newbury Park, CA: SAGE Publications, 2000.

[166] Seidel, R. and Aragon, C., "Randomized search trees," *Algorithmica*, vol. 16, pp. 464–497, 1996.

[167] Shannon, P., Markiel, A., Ozier, O., Baliga, N., Wang, J., Ramage, D., Amin, N., Schwikowski, B., and Ideker, T., "Cytoscape: a software environment for integrated models of biomolecular interaction networks.," *Genome Research*, vol. 13, no. 11, pp. 2498–2504, 2003.

[168] Shi, H. and Spencer, T., "Time-work tradeoffs of the single-source shortest paths problem," *Journal of Algorithms*, vol. 30, no. 1, pp. 19–32, 1999.

[169] Shimbel, A., "Structural parameters of communication networks," *Bulletin of Mathematical Biophysics*, vol. 15, pp. 501–507, 1953.

[170] Singh, B. and Gupte, N., "Congestion and decongestion in a communication network," *Physical Review E*, vol. 71, no. 5, p. 055103, 2005.

[171] Sole, R., Pastor-Satorras, R., Smith, E., and Kepler, T., "A model for large-scale proteome evolution," *Advances in Complex Systems*, vol. 5, no. 1, pp. 43–54, 2002.

[172] Tarjan, R. and Werneck, R., "Dynamic trees in practice," in *Proc. 6th Workshop on Experimental Algorithms (WEA 2007)*, Lecture Notes in Computer Science, (Rome, Italy), pp. 80–93, Springer-Verlag, June 2007.

[173] Thorup, M., "Undirected single-source shortest paths with positive integer weights in linear time," *Journal of the ACM*, vol. 46, no. 3, pp. 362–394, 1999.

[174] Tong, A. *et al.*, "Global mapping of the yeast genetic interaction network," *Science*, vol. 303, no. 5659, pp. 808–813, 2004.

[175] Träff, J. L., "An experimental comparison of two distributed single-source shortest path algorithms," *Parallel Computing*, vol. 21, no. 9, pp. 1505–1532, 1995.

[176] Uetz, P., Giot, L., Cagney, G., Mansfield, T. A., Judson, R. S., Knight, J. R., Lockshon, D., Narayan, V., Srinivasan, M., Pochart, P., Qureshi-Emili, A., Li, Y., Godwin, B., Conover, D., Kalbfleisch, T., Vijayadamodar, G., Yang, M., Johnston, M., Fields, S., and Rothberg, J., "A comprehensive analysis of protein-protein interactions in saccharomyces cerevisiae," *Nature*, vol. 403, pp. 623–627, 2000.

[177] Ullman, J. and Yannakakis, M., "High-probability parallel transitive closure algorithms," in *Proc. 2nd Annual Symp. on Parallel Algorithms and Architectures (SPAA 1990)*, (Crete, Greece), pp. 200–209, ACM, July 1990.

[178] University of Virginia, "Oracle of Bacon." www.oracleofbacon.org, 2008.

[179] Vazquez, A., Flammini, A., Maritan, A., and Vespignani, A., "Global protein function prediction in protein-protein interaction networks," *Nature Biotechnology*, vol. 21, no. 6, pp. 697–700, 2003.

[180] Vazquez, A., Flammini, A., Maritan, A., and Vespignani, A., "Modeling of protein interaction networks," *Complexus*, vol. 1, pp. 38–44, 2003.

[181] Villa, O., Scarpazza, D., Petrini, F., and Peinador, J., "Challenges in mapping graph algorithms on advanced multi-core processors," in *Proc. 21st Intl. Parallel and Distributed Processsing Symp. (IPDPS 2007)*, (Long Beach, CA), March 2007.

[182] Wasserman, S. and Faust, K., *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[183] Watts, D. and Strogatz, S., "Collective dynamics of small world networks," *Nature*, vol. 393, pp. 440–442, 1998.

[184] Werneck, R., *Design and Analysis of Data Structures for Dynamic Trees*. PhD thesis, Princeton University, Princeton, June 2006.

[185] Wuchty, S. and Almaas, E., "Peeling the yeast protein network," *Proteomics*, vol. 5, no. 2, pp. 444–449, 2005.

[186] Yoo, A., Chow, E., Henderson, K., McLendon, W., Hendrickson, B., and Çatalyürek, Ü. V., "A scalable distributed parallel breadth-first search algorithm on Bluegene/L," in *Proc. Supercomputing (SC 2005)*, (Seattle, WA), Nov. 2005.

[187] Zanette, D., "Critical behavior of propagation on small-world networks," *Physical Review E*, vol. 64, no. 5, p. 050901, 2001.

[188] Zaroliagis, C., "Implementations and experimental studies of dynamic graph algorithms," in *Experimental algorithmics: from algorithm design to robust and efficient software*, pp. 229–278, New York, NY: Springer-Verlag, 2002.

[189] Zhan, F. and Noon, C., "Shortest path algorithms: an evaluation using real road networks," *Transportation Science*, vol. 32, pp. 65–73, 1998.

VITA

Kamesh Madduri was born in Rockville, Maryland on January 26, 1983. He received his undergraduate degree in 2004 from the Indian Institute of Technology Madras. His research interests include high performance computing, parallel algorithms, and software support for large-scale data analysis and scientific applications. Kamesh was supported by an NASA graduate student fellowship from 2006-08, has received the 2008 Outstanding Graduate Research Assistant award from the College of Computing at Georgia Tech, and awarded honorable mentions from the ACM/IEEE High Performance Computing PhD fellowship committee in 2007, and the NSF graduate research fellowship program in 2005. His dissertation research has been selected for presentation in doctoral colloquia events at Supercomputing 2007 and IPDPS 2008, and he received the best poster award at the IPDPS 2008 TCPP Ph.D. forum.