31fd421 Fri Jun 16 15:25:48 2017 -0700

CHARACTERIZING AND IMPROVING GRAPH ALGORITHM PERFORMANCE ON MULTICORE SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Nicole Celeste Rodia
June 2017

Abstract

Due to the increasing popularity and scale of graph datasets for domains as diverse as machine learning, data mining, and scientific computing, graph analysis is an important and growing area of research. Graph algorithms perform poorly on modern computer systems, thus we study the reasons behind this lack of performance and demonstrate software and hardware techniques that improve performance. We focus on real-world graphs, which are used in many of today's mainstream graph analytics applications, and show how their properties result in different execution characteristics.

We analyze the performance characteristics of a core set of graph analysis algorithms across several informatics, physical, and synthetic graph datasets using a multicore microarchitectural simulator. We asses the scalability of these applications and their sensitivity to cache size and main memory latency and bandwidth. We also evaluate their input sensitivity to various types of graph datasets. Despite a moderate amount of inherent data locality, modern caching techniques show limited effectiveness for graph algorithms, which are communication-bound. We find that cache miss rates are an unreliable metric for data locality because they are heavily influenced by dataset size, cache size, and replacement policy. Thus, we use cache-independent locality analysis techniques, including reuse distance and a probability-based locality score, to analyze data locality in graph algorithms.

Based on our analysis of data locality, we find that LRU-based cache replacement policies do not provide good performance for graph algorithm data access patterns. Further, we show that data access patterns correlate with algorithm characteristics, graph dataset structure, and vertex degree. These insights indicate that utilization of algorithm- and dataset-specific information paired with small changes to the memory hierarchy could improve graph analysis algorithm performance.

One fundamental graph analysis algorithm used in many scientific and engineering domains is detecting strongly connected components (SCCs) in a directed graph. Traditional approaches in parallel SCC detection show limited performance and poor scaling behavior when applied to large real-world graph instances. We investigate the shortcomings of the conventional approach and propose a series of extensions that account for the fundamental properties of real-world graphs, particularly the small-world property. Our scalable implementation offers excellent performance on diverse small-world graphs resulting in a 5.01x to 29.41x

parallel speedup over an optimal sequential algorithm on 16 cores and 32 hardware threads.

We propose a new cache replacement policy based on our observations of data locality in graph algorithms.

Acknowledgments

I would like to thank...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, interest in graph analytics has exploded across many domains, including machine learning, data mining, and scientific computation. Increasing dataset size coupled with a transition from graphs based on physical systems to informatics graphs has brought the performance challenges of graph analytics to the forefront. These large-scale graph datasets come from online social networks [41], genomics [56], computational biology [27], and the Web [19].

The rise of big data analytics has contributed to the increasing popularity and scale of graph datasets, positioning graph analysis as an important and growing research area. Real-world graphs, derived from data such as social networks, web page links, and paper citations, possess fundamental properties that differ from traditional graphs like trees or meshes, resulting in different execution characteristics. At the same time, modern multicore systems have been scaling to higher core counts, now with tens of complex cores per socket. At first glance, it would seem that graph algorithms can leverage this large number of cores to quickly process large datasets. On multicore systems, however, graph algorithms are typically inefficient and slow. This poor performance is due to several factors, including irregular memory access patterns, load imbalance, and high communication-to-computation ratio.

1.1 Challenges in Graph Analysis

Due to the increasing scale of network analysis problems, with graphs containing millions or billions of vertices and edges, parallel and distributed graph algorithms are essential for effective analysis of these large datasets. Graph applications have significant potential for data-level parallelism across vertices or edges, but algorithm and dataset characteristics make it difficult to achieve high performance and efficiency on real machines.

In addition, graph algorithms in general have several inherent characteristics which make achieving peak

throughput on multicore machines difficult, including data-driven computation, irregular data structure, poor locality, and low computation-to-communication ratio [48]. Data-dependent and spatially-dispersed memory access patterns make traditional memory hierarchies and prefetching techniques less effective for graph applications in comparison to regular algorithms. Combined with low computation per memory access, the result is that runtime is dominated by access to main memory [31].

These algorithms can be executed on modern multicore systems, which support dozens of cores and more than a terabyte of main memory. The data parallelism inherent in graph analytics maps well to the large number of cores in these systems. While compute clusters are also used for large-scale graph processing, due to the communication-intensive nature of graph algorithms, execution time is much faster when graph analysis is run in-memory on a single machine. Additionally, the monetary cost of a multi-machine cluster can be much higher than a single multicore machine. In fact, several graph analytics frameworks, such as GraphChi [43], Ligra [68], X-Stream [62], and Green-Marl [33], target single-machine multicore systems. For these reasons, we focus on computing graph analytics on large multiprocessor systems.

As multicore systems become more capable with increasing core counts and memory capacity, their use for graph analytics will continue to grow. Graph algorithms perform poorly on traditional parallel architectures [48]; thus, to improve performance and efficiency, it is critical to understand the interaction between graph analytics and multicore CPUs and differences from regular parallel algorithms. This knowledge can provide insights for both graph analytics application developers and hardware designers who aim to improve graph analytics performance and efficiency on multicore systems.

we examine the characteristics of a set of optimized graph analysis algorithms and analyze their performance on a shared-memory multicore architecture, focusing on data access patterns and locality. We use architectural simulation to evaluate five algorithms and eight input graphs with various properties. To inform precise claims about data locality in graph algorithms, we perform a thorough quantitative analysis of data locality, memory access patterns, and cache performance. We aim to formalize the execution characteristics of parallel graph algorithms and provide insights for optimizing the performance and efficiency of graph analytics. To improve graph algorithm performance, we present a novel parallel strongly-connected components algorithm and a new cache replacement policy which accounts for data locality patterns.

1.2 Contributions

The goal of this work is to understand the behavior of graph algorithms on multicore architectures, and improve their performance using both software and hardware approaches. In the first part of the talk, I will present a novel algorithm for parallel detection of strongly connected components (SCC) in a directed graph that capitalizes on real-world graph properties to increase parallelism. In the second part of the talk, I will describe an analysis of graph algorithm performance characteristics and memory access patterns on

multicore systems. I use cache-independent locality analysis techniques, including reuse distance and a probability-based locality score, to analyze data locality in graph algorithms. I will show how these results lead to a data cache replacement policy that leverages properties of individual graph algorithms to improve both cache hit rate and overall application performance.

1.3 Thesis Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we provide background on graphs. We review terminology used to describe graphs and their properties, and highlight the properties of real-world graphs and their implications for algorithms. Finally, we describe the graph algorithms and graph datasets used to evaluate them in this work.

To throughly understand graph algorithm performance, we characterize a core set of parallel graph algorithms in Chapter 3. We use multicore simulation to understand the scalability, caching, and memory behavior of the algorithms. Further, we analyze data locality of each algorithm and of the data structures within each algorithm. We use this information to predict the locality and potential for reuse of specific data during execution. We analyze the effect of different algorithms and datasets on the characterization findings. To provide context for these findings, we make comparisons with regular algorithms from the PARSEC parallel benchmark suite.

Leveraging insights about the properties of graphs and the behavior of graph algorithms, we introduce both software and hardware solutions to improve performance. In Chapter 4, we present a new parallel algorithm for Strongly Connected Components, which is a set of three extensions to the existing parallel FW-BW-Trim algorithm. These improvements results in a 5x to 29x parallel speedup over an optimal sequential algorithm. In Chapter 5, we introduce a new cache replacement policy based on the data locality findings from Chapter 3. We implement thi The replacement policy improves performance over LRU by 5%. [TODO: update comparison nums]

We conclude in Chapter 6 with a summary of our contributions and provide suggestions for future work based on our findings.

[TODO: address multiple authorship and self-citation]

Chapter 2

Graph Background

In this chapter, we describe the graph abstraction as context for the remainder of the work. We review the terminology used to describe graphs and their relevant properties. We also describe the input graphs and graph kernels used in our experiments.

2.1 Graph Terminology

To model copmlex systems, such as those found in social relationships, the Internet, the economy, biology, and other domains, researchers use network models. In turn, the graph abstraction uses a graph data structure to represent network-connected data. The vertices in the graph represent entities of interest and the edges represent the connections between them. For example, in a social network, the vertices (also called nodes) correspond to people, and the edges correspond to friendship relationships between people. These networks are then analyzed using a variety of algorithms based in graph theory.

Formally, a *graph* $G(V, E)$ is a structure that contains a set of vertices V and a set of edges E. An *edge (u,v)* connects the pair of vertices *u* and *v*. Edges can be directed or undirected: a *directed egde (u,v)* is a connection from *u* to *v*, while an *undirected edge* represents a bidirectional connection. In the social network example, a directed edge represents a one-way follower relationship and an undirected edge represents a mutual friend relationship. An *undirected graph* is composed exclusively of undirected edges; if a graph contains any directed edges, then it is a *directed graph*.

The *degree* of a vertex is equal to the number of edge connections it has to other vertices. The *degree distribution* is the probability distribution of the vertex degrees over all of the vertices in the graph. The *distance* between two vertices is equal to the fewest number of hops between them. The *diameter* of a graph is the maximum shortest distance between any pair of vertices.

A *strongly connected component (SCC)* of a directed graph, used in our parallel strongly connected

components algorithm in Chapter 4, is a maximal subgraph in which there exists a path between any two vertices in the subgraph.

2.2 Graph Properties

The properties of graph datasets can significantly impact graph algorithm execution patterns and performance. Graphs commonly used for analysis fall into two distinct classes: informatics graphs and meshes. Graphs representing physical networks or structures, such as road networks or finite-element meshes, typically have large diameters and a uniform distribution of small degrees. Thus, these graphs can be partitioned effectively for parallel execution; most communication is local and the amount of work per vertex is similar across vertices. These properties simplify load balancing, and are typical of datasets used in traditional graph processing, such as path-finding in road networks or finite element analysis.

On the other hand, informatics graphs, like social networks and web-link networks, typically exhibit both the small-world and scale-free properties. The *small-world property* means that the graph diameter is very small: $\mathrm{O}(\log\ n)$, where *n* is the number of vertices [71]. The *scale-free property* states that the degree distribution of the vertices follows a power law; many vertices have only a few edges and a few vertices have a very large number of edges [10]. The degree distribution of the LiveJournal online social network in Figure 2.1 demonstrates the scale-free property.

Graphs representing physical networks or structures, such as road networks or finite-element meshes, have very large diameters and a uniform distribution of small degrees. Thus, these graphs can be partitioned effectively for parallel execution; most communication is local and the amount of work per vertex is similar across vertices. These properties simplify load balancing, and are typical of datasets used in traditional graph processing, such as path-finding in road networks or finite element analysis.

Throughout this section, we focus on the case of the LiveJournal graph instance for the sake of explanation. However, in our experiments, we introduce additional real-world graph instances (Section 2.3).

Additionally, in such real-world graphs there exists one giant SCC whose size is $\mathrm{O}(N)$, where N is the number of nodes in the graph [19]. The remaining SCCs are small-sized, and the distribution of SCC size is skewed such that tiny-sized SCCs are much more frequent than large-sized ones [41].

As an illustrative example, Figure 4.2 shows a histogram of the SCC sizes in a real-world graph instance, which is the link relationship of a blog sphere named LiveJournal [47]. This figure shows two aforementioned characteristics of real-world graph SCC structure: the existence of a single giant SCC and the power-law distribution of SCC sizes. The size of the largest SCC (3,828,682) has the same order as the number of nodes in the graph (4,847,571), and the graph has the same order of size-1 SCCs (947,776). The large number of size-1 SCCs explains why the simple Trim step is so effective for SCC detection – it very quickly identifies size-1 SCCs, which are most prevalent in real-world graph instances.

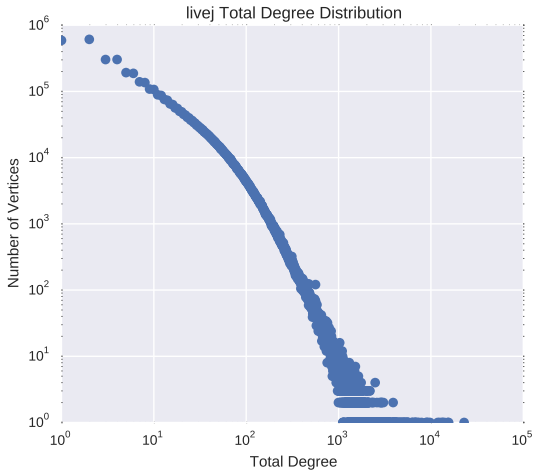


Figure 2.1: The total degree distribution of the LiveJournal online social network follows a power law (scale-free property).

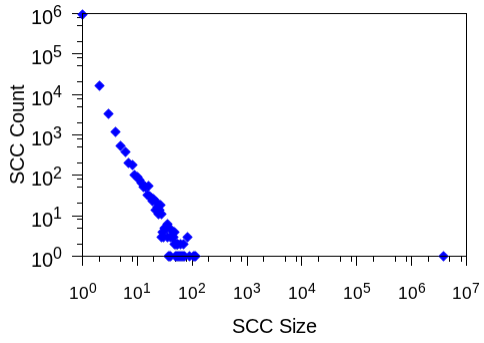


Figure 2.2: Distribution of SCC sizes in the LiveJournal network.

The properties of informatics graphs reduce the efficiency of parallel algorithms. Small-world graphs are often difficult to partition – the 'short-cut' edges that connect otherwise disparate regions of the graph make it difficult to partition the graph with few edge crossings, or a small cut. This leads to load imbalance and additional communication overhead. The scale-free property makes load-balancing difficult. Since the computation per vertex is often proportional to the vertex's degree, high degree vertices do more work.

Consequently, analysis of informatics graphs typically reaches only a small percentage of peak performance on modern shared-memory multicore systems.

2.3 Graph Datasets in this Work

We consider eight graph datasets, which are used as inputs to the algorithms described in Section 2.4. These include both real-world and synthetic graphs of different types, such as informatics and physical networks. Table 4.1 describes these graphs and their basic properties. The datasets were acquired from public repositories [47, 3, 1], and RandomGnm was generated using the GTgraph software [49]. The graph instances are large enough to parallelize and tax the memory hierarchy, but can fit into a single machine's memory and be simulated in manageable time. edges due to long simulation run times.

We focus on real-world informatics graphs, which exhibit the small-world and scale-free properties (Section 2.2). Examples of such graphs are social networks and web hyperlink graphs. We include four graphs in this category, with a range of sizes and average degrees.

For comparison, we include two datasets representing physical networks, which do not have the small-world or scale-free properties: a road network (Germanroad) and a finite element mesh (Ferrari). They have diameters in the thousands, and average degrees no more than six – essentially the opposite of the informatics graphs.

We also include two synthetic networks. The G(n,m) uniform random graph (RandomGnm) is constructed by adding edges between randomly chosen pairs of vertices; duplicate edges are removed. The RMATg500 graph is created using the R-MAT generator, which gives the graph similar properties to real-world networks [21], such as a power-law degree distribution and small diameter.

Name	Graph Description	Type	Vertices	Edges	Degree	Diameter	Max. SCC Size	Size (MB)	Directed
Flickr	Flickr online social network [53]	Social	2,302,925	33,140,018	28.8	28	1,605,184	136	Y
Livej	LiveJournal online social network [7, 46]	Social	4,848,571	68,993,773	28.5	22	3,828,682	282	Y
Baidu	Links in Baidu Chinese online encyclopedia [55]	Web	2,141,300	17,794,839	16.6	20	609,905	77	Y
Orkut*	Orkut online social network [76]	Social	3,072,441	117,185,083	76.3	11	2,963,928	906	Y/N*
Wiki	Links in English Wikipedia [6]	Web	15,172,740	131,166,252	17.2	376	4,736,008	555	Y
Friend*	Friendster online social network [76]	Social	124,836,180	1,806,067,135		25	46,941,703		Y
Twitter	Twitter online follower network [42]	Social	41,652,230	1,468,365,182	70.5	23	16,518,948	5837	Y
Patents	Citation among US Patents [45]	Citation	3,774,768	16,518,948		22	1		Y
CA-road*	California road network [46]	Road	1,965,206	5,533,214		850	1,168,580		Y
Germanroad	German road network [9]	Road	11,548,845	12,369,181	2.1	6,322		139	N
Ferrari	Ferrari 333SP finite element (FE) mesh [9]	Mesh	3,712,815	11,108,633	6.0	1,294		99	N
RandomGnm	Uniform random synthetic graph [49]	Uniform	10,000,000	79,999,969	16.0	14		344	Y
RMATg500	R-MAT synthetic graph, Graph500 benchmark [9]	R-MAT	1,048,576	44,619,402	85.1	7		345	N

Table 2.1: Real-world and synthetic graph datasets used for experiments. Dataset file sizes are in the Green-Marl binary format. Degree is the average degree. For undirected graphs, we list the number of undirected edges, though our implementation represents undirected edges using two directed edges, one in each direction.

2.4 Graph Algorithms in this Work

We select five common graph kernels with a variety of characteristics. Their properties are listed in Table 2.2.

- **Breadth-First Search (BFS)** is a graph traversal kernel that explores the vertices in a graph beginning from a root vertex and records their distance from the root. BFS is a foundational component of many graph algorithms, including betweenness centrality and connected components. We implement a level-synchronous, direction-optimizing algorithm [35], [14]. We run BFS from five random vertices.

- **Betweenness Centrality (BC)** describes a vertex's importance in a graph based on how frequently it lies on shortest paths between pairs of vertices, an important metric in social network analysis [59]. A vertex's BC equals the fraction of shortest paths which pass through that vertex, summed over paths between all pairs of vertices. It is computed with a parallel BFS traversal followed by a parallel BFS over the reverse edges [18]. We estimate BC from five random vertices.

- **PageRank (PR)** is an iterative link analysis algorithm that assigns a numerical weight to vertices, indicative of their relative importance. It was originally developed to rank Web pages [58]. Each vertex's PageRank is a function of the PageRank scores of its incoming neighbors. We run the algorithm for 30 iterations.

- **Hubs and Authorities (HITS)** is an iterative algorithm that assigns hub and authority scores to vertices based on a mutually-reinforcing relationship [40]. A good hub points to many authorities, and conversely, a good authority is pointed to by many hubs. HITS employs the same meta-algorithm as SALSA [44], which can be used as the basis of a user recommendation algorithm for online social networks [29]. We run HITS for 30 iterations.

- **Conductance (CDT)** measures the degree that a graph is tightly-knit [23]. It equals the fraction of edges which cross a cut, or partition of a graph into two sections, normalized by the sum of degrees on one side of the cut. The conductance of a graph is the sum of conductances over all possible graph cuts. We compute conductance for four random graph cuts.

- **Connected Components (CC)**

- **Strongly Connected Components (SCC)** computes maximal subgraphs in which there is a path in both directions between any two vertices in the subgraph. SCC is computed on directed graphs. Our parallel implementation [34] is an improvement on the Forward-Backward-Trim algorithm, optimized for small-world graphs.

We implement all of the graph algorithms in Green-Marl, a domain-specific language (DSL) for parallel graph analysis on shared memory multicores [33]. From the user's high-level code, the Green-Marl compiler

Algorithm	Type	Iterative	Result Type
BFS	Traversal	N	Vertex-to-vertex distance
BC	Centrality	N	Vertex property
PR	Link Analysis	Y	Vertex property
HITS	Link Analysis	Y	Vertex property
CDT	Connectivity	N	Graph or cut property
CC	Components	N	Vertex property
SCC	Components	N	Vertex property

Table 2.2: Comparison of graph analysis algorithm properties.

produces an efficient C++ implementation, parallelized using OpenMP. Results are measured during the main loop of each algorithm, excluding initial graph dataset loading and final results reporting.

Chapter 3

Parallel Graph Algorithm Characterization

3.1 Introduction

To understand how to change graph algorithms or parallel architectures to improve graph algorithm performance, we first seek to understand the performance characteristics of the algorithms. This chapter examines the scaling behavior, efficiency, cache and memory behavior, and data locality of a set of graph algorithms using both hardware performance counters and architectural simulation.

This work appears in [TODO: cite papers here], which we have further expanded and discuss in more detail in this dissertation.

The memory wall, or the difference in speed between CPU and main memory clock rates, is a fundamental limiting factor in graph algorithm performance due to the algorithms' data-intensive nature. Modern processors address this performance discrepancy with a hierarchy of on-chip data caches paired with hardware data prefetching. While it is well established that these techniques perform worse on graph algorithms than regular codes, recent literature disagrees on cache miss rates by 10x [4, 15]. Further, some sources claim inherent 'low' or 'lack of' data locality in graph algorithms [4, 48, 31], and others claim 'substantial' locality [15].

To inform precise claims about data locality in graph algorithms, we perform a thorough quantitative analysis of data locality, memory access patterns, and cache performance. We analyze the performance of a set of optimized graph algorithms on a shared-memory multicore architecture, focusing on data access patterns and locality. We use architectural simulation to evaluate five algorithms and eight input graphs with various properties. This paper aims to formalize the execution characteristics of parallel graph algorithms

and provide insights for optimizing the performance and efficiency of graph analytics.

The insights from our analysis are:

- Cache miss rates are heavily influenced by dataset size, cache size, and replacement policy, and thus are a poor indicator of underlying locality.
- Existing cache techniques show limited effectiveness despite moderate data locality.
- Data reuse patterns in graph algorithms are poorly exploited by LRU cache replacement policies.
- Data access patterns correlate with graph dataset structure and vertex degree.

3.2 Related Work

We focus our study of related work on two main areas: characterization of graph algorithms on CPU and GPU platforms, and locality analysis as it relates to graph algorithms.

Two recent studies characterize parallel graph algorithms on CPU. Beamer et al. [15] identify hardware factors that prevent full utilization of memory bandwidth by graph algorithms. They claim that graph algorithms have 'substantial locality' and 'moderately high' cache hits rates, contrary to most other literature. The authors also find that input graph properties affect performance, which we confirm.

The CRONO graph benchmark paper [4] characterizes a suite of graph algorithms using simulation. The find high data sharing and network traffics as the primary bottlenecks. Their results show low cache miss rates and they conclude that memory bandwidth does not limit scaling. Conversely, the authors state that graph algorithms have 'low locality.' CRONO uses small input graphs (most with less than 3 million edges), whereas we use larger graphs in simulation (up to 131 million edges). We hypothesize that small graphs paired with large caches (65 MB total L2) result in the observed low cache miss rates and memory bandwidth utilization.

Additionally, two earlier studies discuss graph algorithm bottlenecks. Hendrickson et al. [31] and Lumsdaine et al. [48] state the need for parallel graph processing due to large data size. They agree that graph algorithms have a high data access to computation ratio. Both works argue for massively multithreaded systems due to poor locality.

Yang and Chien [75] characterize graph algorithm behavior across different scales of algorithms and graph structures on a cluster of machines. They find a 1000-fold variation in graph computation behaviors.

Several works analyze the behavior of graph algorithms on GPU. Burtscher et al. [20] study irregular algorithms, including graph algorithms, on GPU using hardware performance counters and quantify control-flow and memory access regularity. They find that irregularity in graph applications is sensitive to graph type. A follow-up work by O'Neil et al. [57] uses simulation to further investigate algorithm behavior.

Compared to regular algorithms, graph algorithms exhibit lower IPC, higher L1 MPKI, and more stalls due to uncoalesced memory accesses. Notably, graph applications were more sensitive to interconnect (cache) bandwidth and latency than that of DRAM.

Via GPU performance counters and simulation, Xu et al. [74] identify the main graph algorithm bottlenecks as synchronization with the CPU and memory latency. For the Pannotia GPU graph benchmark suite, Che et al. [22] identify the main performance challenges as branch and memory divergence, load imbalance, and variation in parallelism over the execution, and find diverse behaviors across workloads. Wu et al. [73] analyze graph algorithm primitives on GPU across programming models. They conclude that graph topology significantly effects performance; particularly, graphs with a larger traversal frontier (small diameter) and vertex degree perform better.

We examine work on modeling and analysis of temporal and spatial locality as it applies to graph algorithms. The concept of reuse (or LRU stack) distance was first introduced by Mattson et al. [50]. Jiang et al. [38] introduce concurrent reuse distance, which extends reuse distance to multithreaded workloads on shared-memory multicore machines. Wu et al. [72] apply concurrent and private reuse distance to study the scalability of multicore cache hierarchies.

Gu et al. [28] propose a model for spatial locality based on the change in reuse distance as a function of data block size. Gupta et al. [30] quantify spatial and temporal locality as a conditional probability and visualize locality as a function of near future window size and neighborhood size. Yuan et al. [77] apply locality modeling to graph traversal and introduce vertex distance, with which they predict BFS reuse distance and number of levels on synthetic graphs.

A common theme across the literature is the diversity of behavior across graph algorithms and datasets. To capture variation in behavior, we characterize diverse algorithms and datasets to produce quality characterization results. The range in cache miss rates indicates that they can be unreliable for characterization due to dependence on input graph and cache size. While previous work has presented cache miss rates and reuse distance on synthetic graphs, this paper provides a quantitative analysis of locality and its sources on large, real-world graphs.

3.3 Methdology

We select five common graph kernels with a variety of characteristics. Their properties are listed in Table 2.2. We implement all of the graph algorithms in Green-Marl, a domain-specific language (DSL) for parallel graph analysis on shared memory multicores [33]. From the user's high-level code, the Green-Marl compiler produces an efficient C++ implementation, parallelized using OpenMP. Results are measured during the main loop of each algorithm, excluding initial graph dataset loading and final results reporting.

3.3.1 Zsim Architectural Simulator

For simulations, we use the zsim microarchitectural simulator [64]. Zsim models a multiprocessor system comprised of processing cores, each with a single execution thread, and a multi-level cache hierarchy connected to a main memory model. Zsim leverages dynamic binary translation and instruction-driven timing models to achieve fast simulation times. Since memory access behavior dominates graph algorithm performance, we can evaluate real-world datasets, which provides more realistic results.

The architectural parameters for our 16-core simulator configuration appears in Table 3.1. For applicable experiments, we note variation to these baseline system parameters. We use zsim's out-of-order quad-issue core model, based on Intel's Westmere architecture. Where specified, we instead run zsim with a simple core model, with core IPC equal to one. We use the MD1 memory model, which is based on a nominal memory latency and maximum memory bandwidth. To model queueing delay, as load on the memory increases, the latency experienced by each access increases.

Parameter	Value
Num. Cores	16 @ 3.0 GHz
L1-I Cache per core	32 KB, 4-way assoc., 4 cycles
L1-D Cache per core	32 KB, 8-way assoc., 4 cycles
L2 Cache shared per 2 cores	512 KB, 16-way assoc., 10 cycles
L3 Cache shared	16 MB, 16-way assoc., 38 cycles
Cache Line Size (bytes)	64
Cache Configuration	Inclusive, MESI coherence
DRAM Bandwidth	12.8 GB/sec
DRAM Latency	175 cycles
Num. Memory Controllers	1

Table 3.1: Zsim architectural parameters.

The cache and memory latencies are estimated based on measurements of the Nehalem architecture [54]. We use the lowest latency values for cache and memory, which makes the performance of our simulated memory hierarchy optimistic compared with a real system with non-uniform cache and memory access effects. Thus, on real machines, we expect worse cache performance, which makes cache optimizations for graph algorithms even more critical.

We run a recent version of Zsim from the GitHub repository [63], instrumented with additional performance counters, and Pin library version 2.14.

3.3.2 Reuse Distance and Locality Algorithms

We use two algorithms for data locality analysis. One is the reuse distance algorithm, also known as LRU stack distance [50], which measures temporal locality. The other is a probability-based locality analysis [30],

which accounts for both spatial and temporal locality.

To create the memory traces used in these analyses, we generate a memory trace for each thread's loads and stores with zsim using the simple core model. We interleave all threads' accesses into one stream, with ordering based on the execution cycle of the accesses. Due to the length of the memory traces produced from our graph algorithms, we analyze locality for only one "iteration" of each algorithm: one start node for BFS and BC, one iteration for PR and HITS, and one graph cut for CDT. The address granularity for accesses is 4 bytes, which independently captures the reuse of most data elements, including `int`, `float`, `long`, and `double`. For data structure-specific reuse distance, we compute the reuse distances from the full trace, then filter by the addresses specific to the data structure.

Our reuse distance algorithm implements in C++ the Bennett and Kruskal partial sum optimization [17], which reduces the computational complexity from O $\left(n^2\right)$ to O $(log(n))$, where *n* is the length of the memory trace. We use locality probability analysis code provided by its author, which is implemented in C [30]. Our code is compiled with gcc/g++ version 4.8.2 with the -O3 option, and run on Ubuntu 14.04.

3.4 Validation of Simulator Results

To validate the results gathered from the zsim simulator, we compare them with hardware performance counter values from a real x86-64 multicore processor. In additon to validation published in [64], we compare zsim's performance numbers with performance counter results from Intel PCM.

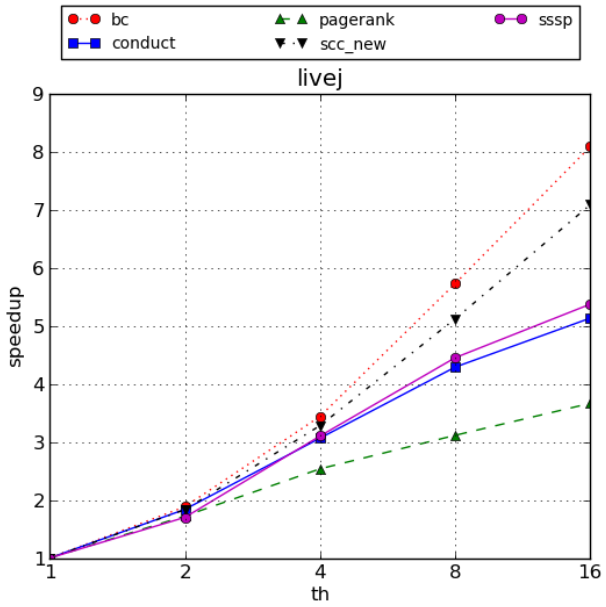
3.5 Scalability and Efficiency

Data parallelism in graph algorithms provides potential for significant scalability, however, it is limited by irregularity in control flow and data access. Figure 3.1(a) shows the parallel speedup over one thread for the LiveJournal dataset up to 16 threads. The scalability varies between algorithms, with a range from 3.6x for PageRank to 8.1x for Betweenness Centrality.

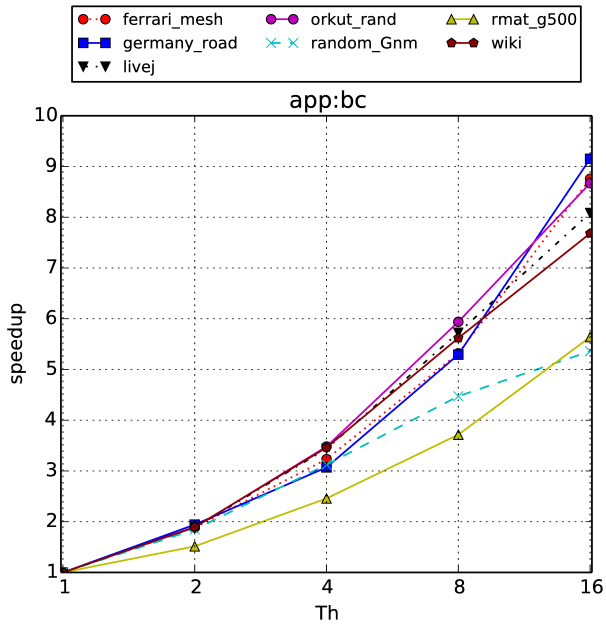
Figure 3.1(b) shows speedup for the same simulated system, but on the Betweenness Centrality (BC) algorithm over the full set of input graphs. The speedup on BC for 16 threads ranges from 5.3x to 9x, less than 60% of the ideal speedup.

To further investigate the parallelism in our graph algorithms, we consider scaling on a 128-core simulated system. Figure 3.2 shows the speedup for BC on this system. The maximum speedup is limited to 48x on 128 cores, only 38% of the ideal speedup.

We assess processor efficiency on graph algorithms using instructions per cycle (IPC). Figure 3.3 shows the IPC as a function of thread count on the 16-core system. The simulated 4-way superscalar cores have an ideal maximum IPC of four. In Figure 3.3(a), the graph applications, with IPC ranging from 0.35 to 0.75 for



(a) LiveJournal dataset



(b) Betweenness Centrality (BC) algorithm

Figure 3.1: Parallel speedup over 1 core execution on 16-core simulated system.

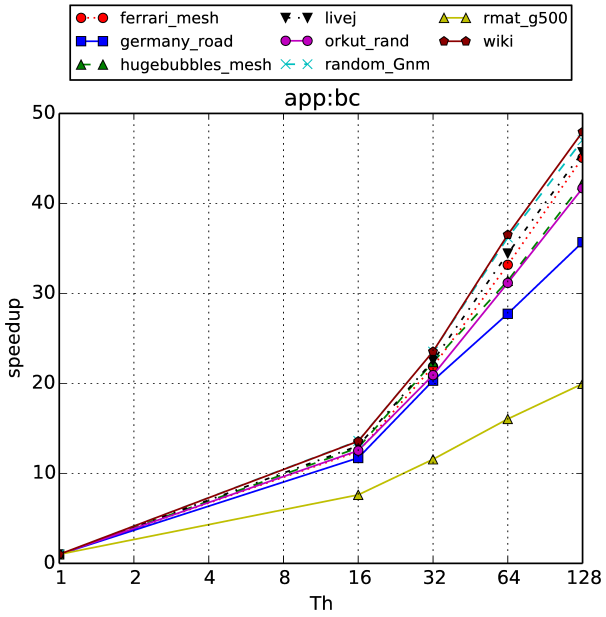


Figure 3.2: Parallel speedup over 1 core execution on 128-core simulated system for the Betweenness Centrality algorithm.

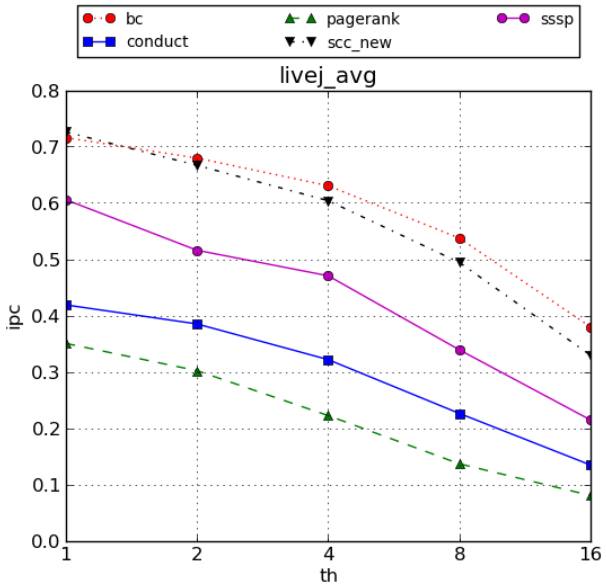
the LiveJournal dataset on one thread, fall far short of the potential core throughput.

Additionally, the IPC decreases 3-4x for higher thread count, which indicates that execution becomes less efficient with increasing number of threads. We attribute this effect primarily to memory bandwidth saturation.

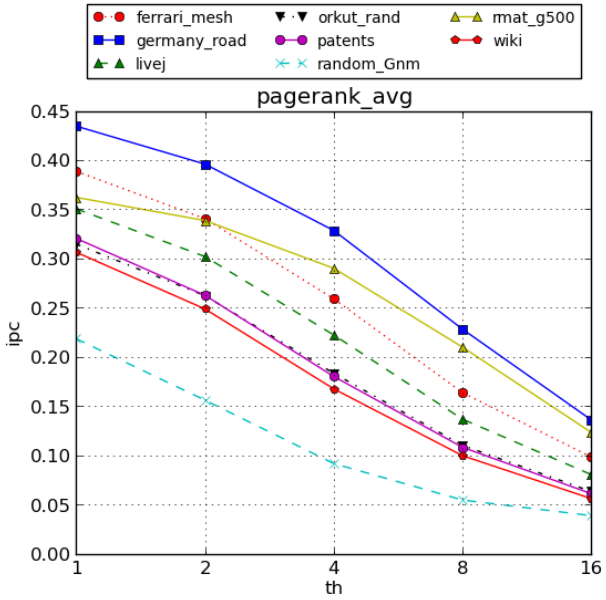
With all but the most embarrassingly parallel workloads, parallel applications suffer from myriad overheads, including communication between threads, load imbalance, synchronization cost, and additional algorithmic complexity.

In our level-synchronous BFS implementation, lack of dynamic parallelism in the application, which leads to load imbalance, is a major contributor to less-than-ideal scaling. The beginning and ending levels of the traversal have little parallelism since there are few nodes in the frontier, while the middle levels have sufficient parallel tasks to distribute to 128 threads.

Since Betweenness Centrality is built on level-synchronous BFS, it exhibits these load imbalance issues as well. For Betweenness Centrality, the small-world graphs, including social and link networks, have the best parallel speedup. This is because the level-synchronous BFS used in our BC implementation can derive significant parallelism through most of the execution for small diameter graphs, but for large diameter graphs, the parallelism is limited since there are fewer active vertices in each level of the traversal. Figure 3.2 illustrates the performance difference between graph datasets that occurs in BC.



(a) LiveJournal dataset



(b) PageRank algorithm

Figure 3.3: IPC versus thread count on 16-core simulated system.

3.6 The Memory Wall

The data-intensive nature of graph algorithms makes the ability to quickly access large amounts of data a fundamental concern for performance. The memory wall causes particular performance difficulties because there is little computation for each data access, which places a heavy load on the memory hierarchy in comparison to the compute resources. Load and store instructions make up about one-third of the total instructions executed, as shown in Figure 3.4. There are an average of three computation instructions for every two memory instructions, which gives few compute operations between memory accesses to hide the many-cycle latency to the last-level cache (LLC) or main memory.

The computational instructions are often integer-based, but certain algorithms perform floating point operations, including BC, HITS, and PR. Most control instructions are tight loops and conditionals. Branch misprediction rates are less than 0.12 for all of our algorithm and dataset combinations. Combined with high memory bandwidth load, instruction fetch is not a bottleneck for generating sufficient outstanding memory accesses to saturate the memory system.

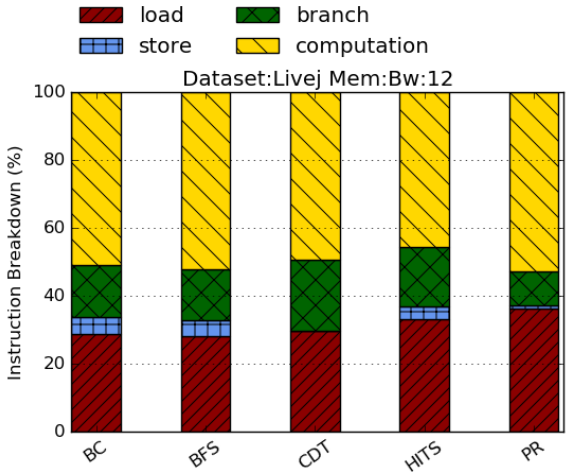
Main memory bandwidth is a significant factor limiting performance and scalability. We plot instructions per cycle (IPC) as a function of memory bandwidth in Figure 3.5. Figure 3.5a demonstrates that the IPC for PR increases by 3-4x for an increase in memory bandwidth of 15x. Since PR iterate through all of the edges in the graph, and read and writes all PageRank values, its IPC continues to increase all the way to 100 GB/sec. In Figure 3.5b, we see the benefit of increased memory bandwidth begin to wane at 50 GB/sec for BFS and BC. These applications also have the highest IPC and lowest LLC miss rates due to data access patterns with higher locality.

Due to a low computation-to-memory access ratio and cache misses (Section **??**), many of the loads executed in an algorithm suffer the latency of an access to main memory. Limited memory bandwidth and high memory latency are the primary causes of the low IPC rates (<1 on a quad-issue core) observed in Figure 3.5.

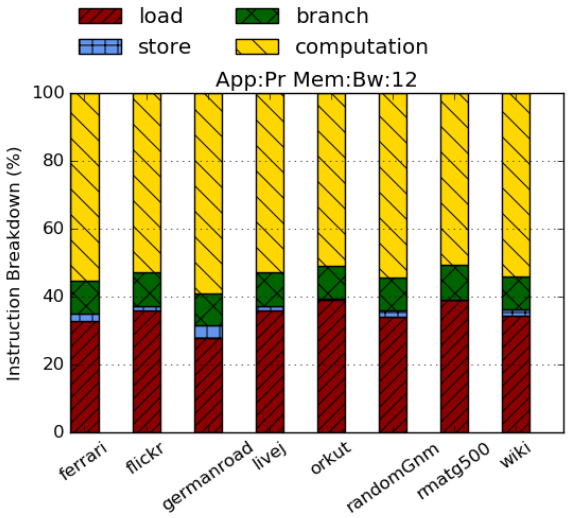
Another factor in the need for high memory bandwidth are scatter-gather memory access patterns that result in inefficient use of the memory system. In the PR algorithm, each vertex reads the PageRank values of its in-neighbors, which are distributed throughout the PageRank value array. Though each PageRank value is an 8-byte `double`, the system must load a full 64-byte cache line, even if only a single PageRank value is accessed before eviction.

Figure 3.6 shows the percentage of 8-byte words accessed in all 64-byte cache lines before the line is evicted. Generally, only one 8-byte word is touched for over 80% of cache lines brought into the cache. The Ferrari and Germanroad datasets have higher cache line utilization because of their smaller sizes, so more data elements can be accessed before a cache line is evicted.

In addition to wasting bandwidth with unused data, modern DRAMs are much slower at random data access than sequential. If all access is random, peak memory bandwidth can only reach 22% of maximum

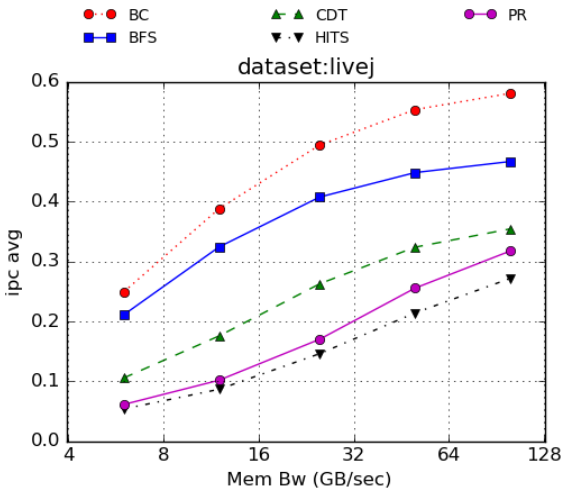


(a) Livej dataset on all algorithms

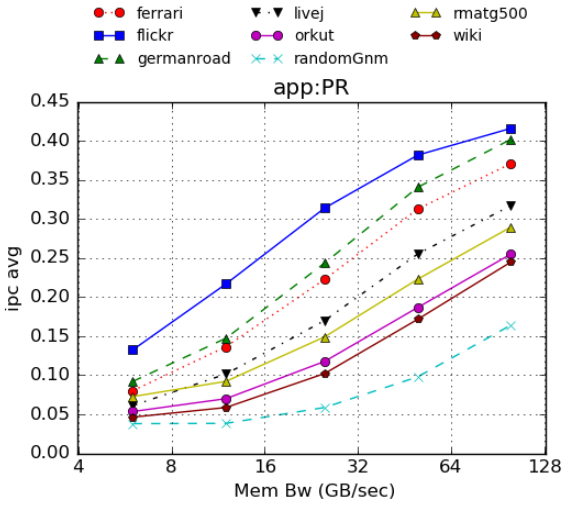


(b) All datasets on PR algorithm

Figure 3.4: Dynamic instruction breakdown.



(a) Livej dataset on all algorithms



(b) All datasets on PR algorithm

Figure 3.5: IPC relative to main memory bandwidth.

sequential bandwidth for one core, or 55% for 16 cores [62]. Since most of the graph algorithm data access patterns are not sequential, this further compounds the issue of limited memory bandwidth.

3.7 Cache Performance

Modern processors partially mitigate the negative performance effects of the memory wall with a multi-level hierarchy of high-capacity data caches. For graph algorithms, accessing gigabytes of data decreases cache effectiveness.

We examine the effect of LLC size on LLC cache miss rates, and find that data cache size significantly affects our graph algorithm performance. Increasing the size of the LLC lowers LLC cache misses per thousand instructions (MPKI) across all of the algorithms and datasets that we evaluated. The effect is largest for the PageRank algorithm, shown in Figure 3.7. We expect this trend because the PageRank algorithm cycles through all of the data structures during each iteration of the algorithm. Thus, a larger LLC can fit a larger portion of these data structures, reducing reads from main memory.

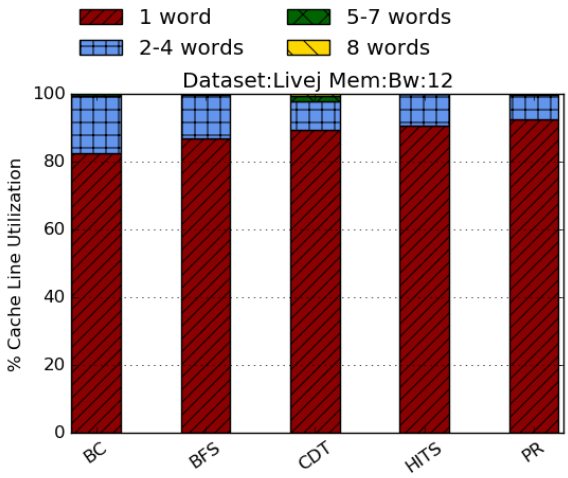
3.8 Data Locality

We seek to understand the sources of locality in graph algorithms to boost performance by maximizing data reuse. We consider both types of data locality: temporal and spatial. Given that a particular memory location is accessed at a point in time, temporal locality is the likelihood that the same location will be accessed again in the future; spatial locality is the likelihood that a nearby location will be accessed in the future.

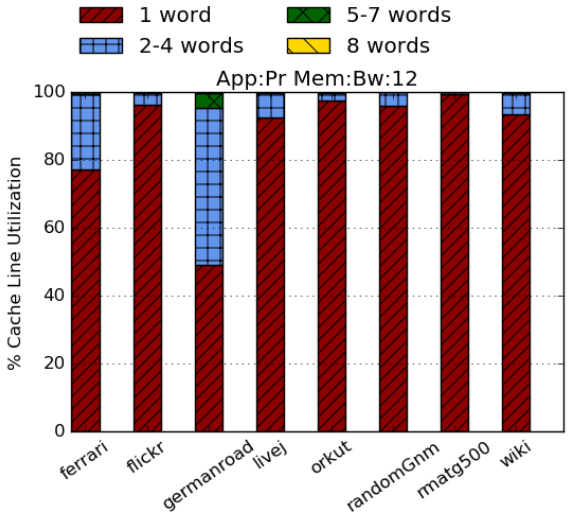
3.8.1 Reuse Distance

Reuse distance analysis is a technique for measuring temporal locality in a memory access stream. For an application memory trace, the reuse distance is defined as the number of distinct data elements accessed between two consecutive accesses of the same element. Measuring reuse distance is advantageous because it is independent of cache configuration and allows analysis of program behavior separately from a given machine or cache design [25]. We examine the reuse distance for a single iteration of each algorithm (see Section 3.3.2).

As a running example, we consider the reuse distance for the Livej dataset on the PR algorithm. Figures 3.8a and 3.8b show the reuse distance probability distribution function (PDF) and cumulative distribution function (CDF) of the Livej dataset on PR, respectively. The PDF shows the percentage of total memory accesses for each reuse distance. The CDF shows the percentage of total accesses which are captured by a given reuse distance. The CDF may not reach 100% if some data is never reused during one iteration of the algorithm.



(a) Livej dataset on all algorithms



(b) All datasets on PR algorithm

Figure 3.6: Percentage of 8B words accessed in a 64B cache line before the line is evicted, over all cache lines in the L1 data cache accessed during execution.

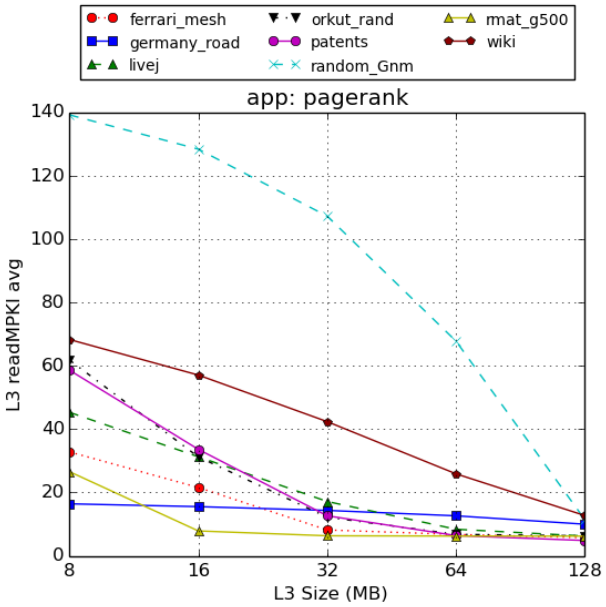


Figure 3.7: LLC (L3) data cache read MPKI relative to LLC size for all datasets on the PR algorithm.

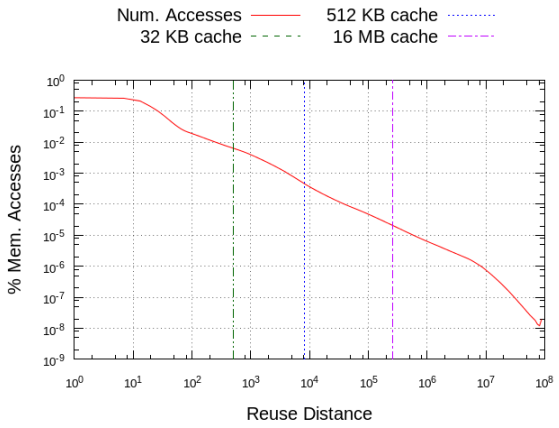
To show how many accesses would be hits in a hypothetical fully associative cache with the least recently used (LRU) replacement policy, we superimpose vertical lines on the histogram corresponding to various cache sizes. Accesses to the right of a cache size line would be cache misses, because too many pieces of data have been brought into the cache since the address's previous reference.

The long reuse distance accesses on the right side of the plots are a poor match for the LRU replacement policy because cached data is likely to become least-recently used and evicted before it has the opportunity to be reused. This suggests that a thrash-resistant replacement policy may improve cache hit rates.

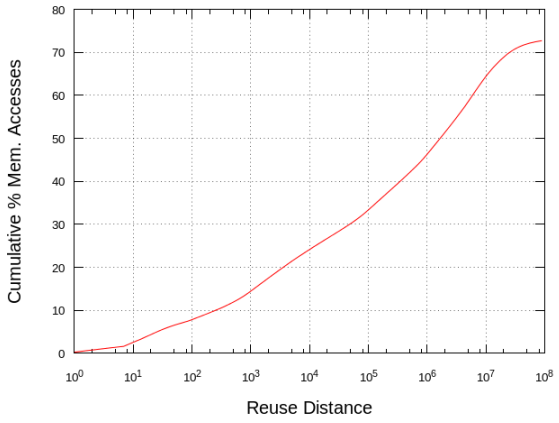
3.8.2 Locality Probability

We formally define the two types of locality as conditional probabilities [30]. We define temporal locality as the following conditional probability: given that an address A is referenced, the probability that A is reaccessed in the near future. We define near future as the next N *unique* addresses in the address stream. Similarly, we define spatial locality as the following conditional probability: given that an address A is referenced, the probability that an address in its neighborhood is accessed in the near future. We say that an address X is in the neighborhood of an address Y if $|X - Y| < K$, where the size of the neighborhood is $2(K - 1)$. Note that by these definitions, temporal locality is equal to spatial locality with a neighborhood size of zero ($K = 1$).

Figures 3.9a and 3.9b show data locality as a function of neighborhood size (spatial locality) and near

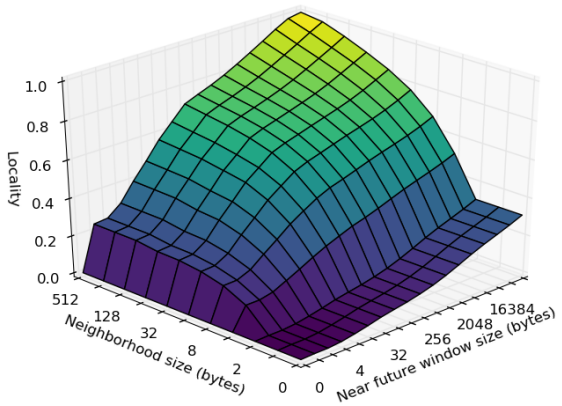


(a) Reuse distance PDF

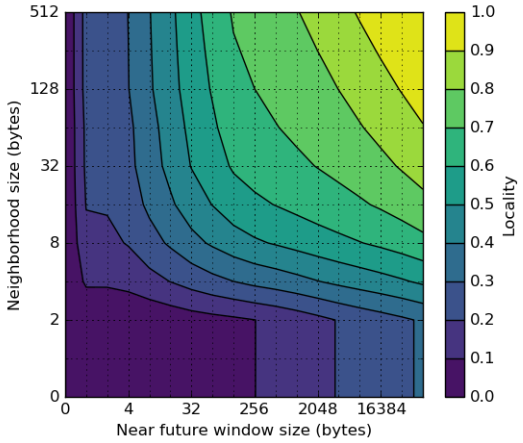


(b) Reuse distance CDF

Figure 3.8: Reuse distance for the Livej dataset on the PageRank algorithm.



(a) Locality score as a 3D surface



(b) Locality score as a 2D contour

Figure 3.9: Locality score as a function of near future window size and cache block size for the Livej dataset on the PR algorithm.

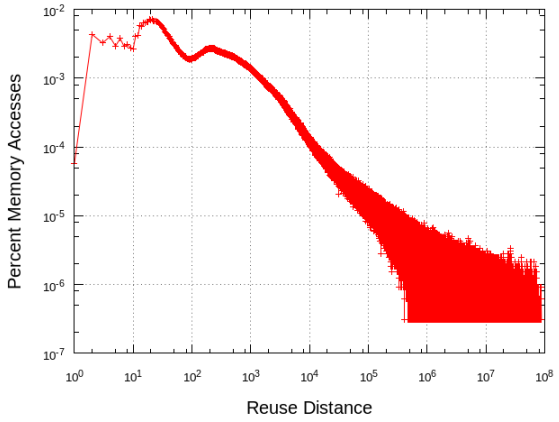


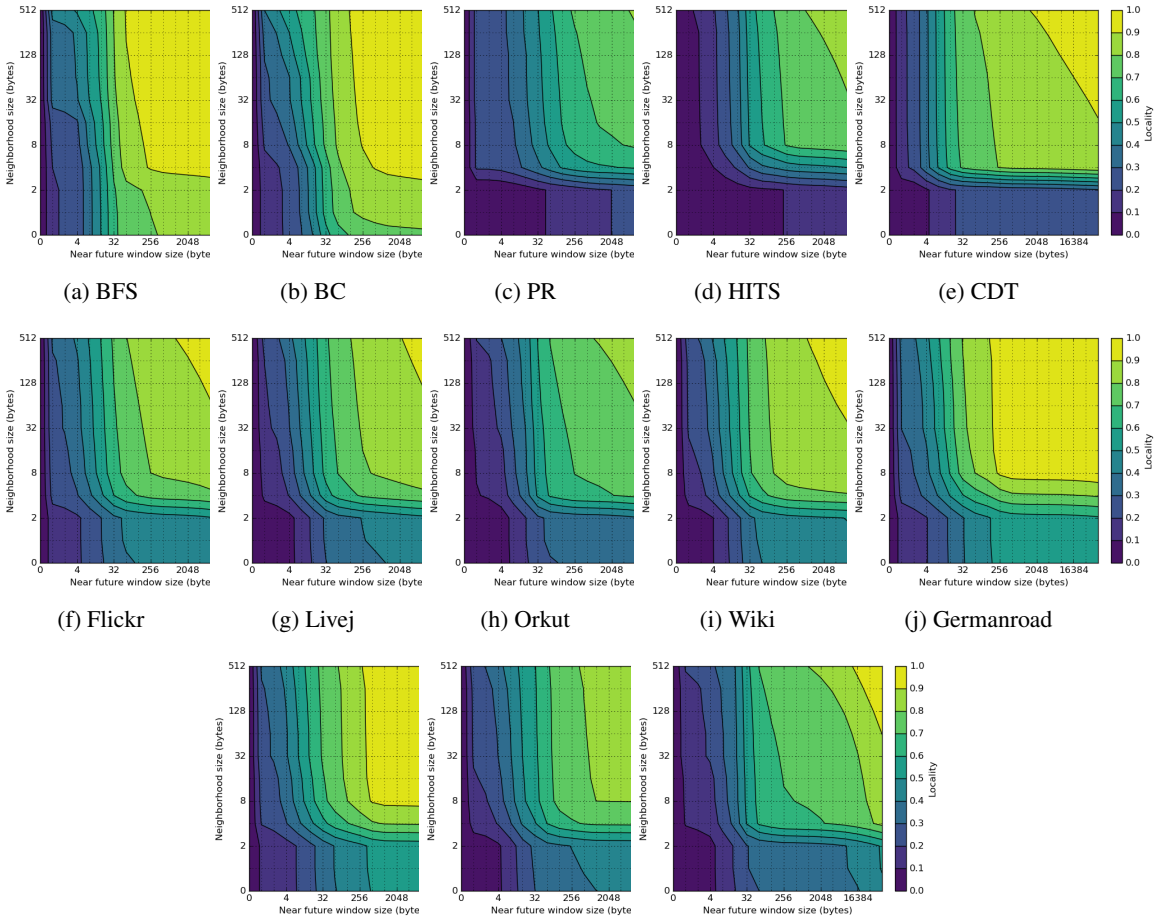
Figure 3.10: Reuse distance PDF of PageRank values array for Livej dataset.

future window size (temporal locality) for the Livej dataset on PR. In Figure 3.9a, slices of constant neighborhood size correspond to the reuse distance CDFs for different cache block sizes. To match the reuse distance plots, which have a 4-byte address mask, we use locality measurements with neighborhood size determined by address mask.

Most of the locality is temporal, shown in Figure 3.9a as an increase in locality with an increase in near future window size. Increasing the neighborhood size has some effect on the overall locality, but it increases slowly once the neighborhood size reaches four bytes, which is the size of most data elements. Similar results are observed for PR on the other datasets, and for HITS and CDT. For BFS and BC, spatial locality has a less significant contribution.

In Figure 3.11, we plot average locality profiles as 2D contours for each algorithm and each dataset. For example, PR (c) is an average the locality scores of each of the eight input graphs run on the PR algorithm; Livej (g) is an average of the Livej input graph run on all five algorithms. The average standard deviation for each locality profile is shown in Table 3.2. There is more variation in locality for different applications than for different datasets on the same application. As such, locality optimizations may require tailoring for each algorithm or class of algorithms, but are more generalizable across datasets.

The size and structure of graph datasets are important factors in data locality. For example, the small degree and uniform structure of the Ferrari and Germanroad graphs provide better temporal locality than scale-free graphs because of the greater potential for reuse when vertices are more tightly-knit.



(a) BFS (b) BC (c) PR (d) HITS (e) CDT

(f) Flickr (g) Livej (h) Orkut (i) Wiki (j) Germanroad

(k) Ferrari (l) RandomGnm (m) Rmatg500

Figure 3.11: Locality score 2D contour plots averaged over algorithms or datasets.

Algorithm	Std. Dev. (10^{-3})
BC	1.55
BFS	6.60
CDT	7.77
HITS	7.08
PR	9.26

Dataset	Std. Dev. (10^{-3})
Ferrari	24.4
Flickr	36.7
Germanroad	21.6
Livej	24.5
Orkut	33.7
RandomGnm	33.7
Rmatg500	34.2
Wiki	22.0

Table 3.2: Average standard deviations of locality scores for algorithms and datasets.

3.9 Whole Program Data Locality

3.10 Per-Data Structure Data Locality

3.11 Locality Prediction

We can leverage knowledge about each algorithm's data access patterns and the structure of graph datasets to predict the locality of data, which could be used to improve cache performance.

By performing a simple hand analysis of the access patterns to the primary data structures in each algorithm, we can predict which data structures are reused within one iteration of the algorithm and whether the reuse distance correlates to vertex in- or out-degree. For each application, each data structure has an access pattern with some consistency across input datasets.

We extend reuse distance analysis to study the access patterns of specific data structures in each algorithm. Figure 3.10 shows the reuse distance PDF for the array that holds the PageRank values for each vertex in the PR algorithm. The shape of the reuse distance distribution is similar to that of reuse distance for the full algorithm.

To fully understand locality at the data structure level, we must examine the inner workings of our algorithms. Green-Marl represents the graph vertices and edges using the compressed sparse row (CSR) adjacency list format. This consists of an array of row pointers of length number of vertices (G.begin) and an array of column indices of length number of vertices (G.node_idx). Additionally, Green-Marl creates a set of reverse graph CSR arrays, which are used in BFS for the bottom-up portion of the direction-optimizing algorithm, in BC for the reverse BFS, and for traversing in-neighbors in PR and HITS.

While we look at the locality of each iteration in isolation, there is of course opportunity for reuse between iterations. Particularly, there is no reuse of the G.node_idx data structure within an iteration. Due to the immense size of this array, today's caches will not be able to maintain its full state, let alone the other data structures needed during execution, within the cache across iterations. The G.node_idx array of our largest dataset, Wiki, is 500 MB, far too large even for the latest on-chip LLCs, which are less than 100 MB.

If column indices were not cached, there would be potential to reuse other data structures between iterations, however these data structures are still tens of megabytes. For our example of Wiki, an array of vertex properties of size `double` (8 bytes), such as the PageRank values, is 116 MB.

Clearly, simply selecting data structures to avoid caching, such as cache hints based on particular instructions (proposed in [4]), will not be sufficient to avoid thrashing, or evicting data with potential for reuse.

We find that there is a correlation between vertex degree and reuse distance of that vertex's properties, which is most pronounced for scale-free graphs because of their large range of degree values. Figure 3.12 shows the relationship between average reuse distance and vertex out-degree for the PageRank data structure.

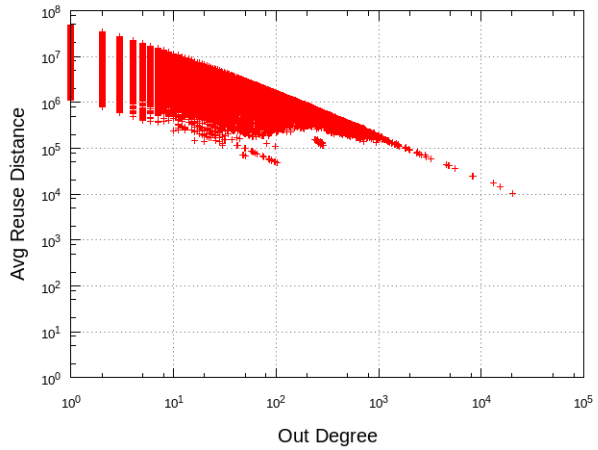


Figure 3.12: Average reuse distance relative to vertex out-degree for PageRank data structure for Livej dataset on PageRank algorithm.

We expect a correlation with out-degree because for all of the in-neighbors of each vertex, one data element is read from the PageRank values array. The more out-neighbors a vertex has (high out-degree), the more likely it is to be an in-neighbor of any vertex, thus the expected correlation.

3.12 Analysis

3.13 Effect of Algorithms and Datasets

3.14 Comparison with Non-graph Algorithms

PARSEC

3.15 Conclusion

This paper presents a performance characterization of several parallel graph analysis algorithms on real-world datasets. Our analysis is based on comprehensive experiments from architectural simulation of shared-memory multicore systems. We demonstrate that existing cache and prefetching techniques show limited effectiveness in spite of available data locality. We find that graph algorithm data reuse patterns correlate with graph dataset structure and vertex degree, but cannot be effectively exploited by LRU cache replacement policies. These insights can be leveraged to improve graph analysis algorithm performance.

Chapter 4

Parallel Strongly-Connected Components

4.1 Introduction

In graph theory, a strongly connected component (SCC) of a directed graph is a maximal subgraph where there exists a path between any two vertices in the subgraph. Since any directed graph can be decomposed into a set of disjoint SCCs, the study of large graphs frequently uses SCC detection of the target graph as a fundamental analysis step. Target *real-world* graphs include the Web graph and social networks [19, 24, 41], and those found in diverse scientific applications, including formal verification [32], reinforcement learning [39], 3D mesh element refinement [51], and complex food web analysis [5].

Tarjan's algorithm [69], the classic sequential method for SCC detection, is an asymptotically optimal linear-time algorithm. Unfortunately, Tarjan's algorithm is difficult to parallelize because it extends the depth-first search (DFS) traversal of the graph, which is inherently sequential [61].

Several studies [26, 51, 12, 11] have investigated parallel or distributed SCC algorithms. Fleischer et al. [26] devised a practical parallel algorithm, the Forward-Backward (FW-BW) algorithm, which motivated further enhancements in following research. The FW-BW algorithm achieves parallelism by partitioning the given graph into three disjoint subgraphs which can be processed independently in a recursive manner. McLendon et al. [51] added a simple extension to this algorithm, the Trim step, which resulted in a significant performance improvement.

Barnat et al. [12] proposed the recursive OBF algorithm to improve the degree of parallelism compared to the original FW-BW algorithm. However, their method [11] did not give a large performance improvement over McLendon et al.'s when applied to real-world graphs with few large-sized SCCs. Barnat et al. [11]

demonstrate a CUDA implementation based on forward reachability that outperforms the sequential Tarjan's algorithm, but concede that none of their implementations on a quad-core system were able to outperform Tarjan's algorithm.

Although these algorithms show a degree of parallel performance in distributed environments, their parallel performance in shared-memory environments is much lower than that of the optimal sequential algorithm, especially when applied to large *real-world* graph instances. As shown in this paper, this is because the characteristics of *real-world* graphs differ substantially from synthetic graphs, such as trees or meshes, for which those algorithms were originally designed. Studies [19, 10, 71] have identified several fundamental characteristics of *real-world* graphs, in particular the *small-world* property (Section **??**).

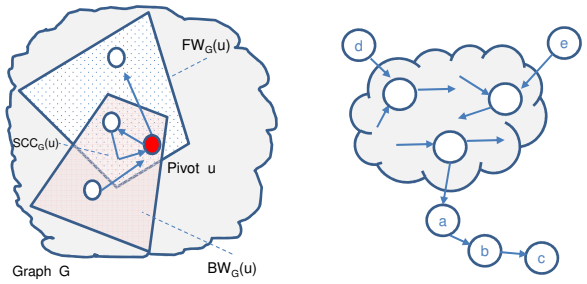
In this paper, we first review McLendon et al.'s parallel algorithm (FW-BW-Trim) before we explain the characteristics of *real-world* graph instances (Section **??**). Next, we introduce our series of extensions to the conventional FW-BW-Trim algorithm, which account for those characteristics (Section 4.4). We discuss issues in implementing these algorithms, which can significantly impact performance (Section 4.5). In our experiments (Section **??**), we run our extended algorithm on a set of *small-world* graph instances and observe the effectiveness of each extension for the characteristics of those instances. Our results show that our methods not only improve the absolute performance of the original FW-BW-Trim algorithm, but also extract a higher degree of parallelism.

Our specific contributions are as follows:

- We identify the performance limitations of the conventional FW-BW-Trim algorithm on large *real-world* graph instances (Sections **??** and 4.3).
- We propose a set of extensions to the conventional algorithm, which consider characteristics of those *real-world* instances, including the *small-world* property (Section 4.4).
- We explain the performance-critical implementation details of the conventional algorithm and extensions (Section **??**).
- We analyze the effect of our extensions with varying *small-world* graph shapes (Section 4.6). To our knowledge, we demonstrate the first parallel SCC algorithm which outperforms Tarjan's algorithm on a shared-memory multiprocessor machine on such graphs.

4.2 Related Work

[TODO: discuss new work since paper was published]



(a) (b)

Figure 4.1: The two main ideas of the conventional FW-BW-Trim algorithm: (a) Forward and Backward reachability and (b) Trimming.

4.3 Conventional FW-BW-Trim Algorithm

In this section, we review FW-BW-Trim, a conventional parallel SCC detection algorithm [51]. The FW-BW-Trim algorithm extends its predecessor, the original FW-BW algorithm [26], by adding the Trim step, which detects size-1 SCCs to improve performance.

The original FW-BW algorithm is based on the observations in Lemma 1 [26]. Given a directed graph G, let $FW_G(i)$ be the subset of vertices in G which are reachable from vertex i. Let $BW_G(i)$ be the subset of vertices in G from which i is reachable.

Lemma 1. *Let $G = (V, E)$ be a directed graph with $i \in V$ a vertex in G. Then $FW_G(i) \cap BW_G(i)$ is a unique SCC in G. Moreover, for every other SCC s in G, either $s \subset FW_G(i) \setminus BW_G(i)$, $s \subset BW_G(i) \setminus FW_G(i)$, or $s \subset V \setminus (FW_G(i) \cup BW_G(i))$.*

Lemma 1 states that from any node i in graph G, $SCC_G(i)$, the unique SCC that contains i, can be identified from the intersection of two sets: the forward reachable set of i and the backward reachable set of i, where we call i the pivot node. Furthermore, the remaining nodes can now be partitioned into three subgraphs (forward reachable only, reverse reachable only, and non-reachable) where each subgraph can be processed independently in a recursive manner. Figure 4.1(a) provides a visual explanation of this idea. The computational complexity of the FW-BW algorithm is $\mathrm{O}\,(n + m)$ for each partition, which detects a single SCC [11].

The parallelism of the FW-BW algorithm comes from its recursive application to each partition. Since there cannot be an SCC that belongs to more than one partition, each partition can be processed independently, in parallel. Furthermore, since each partition produces three additional partitions, it is expected that quickly, there would be sufficient independent tasks to consume all of the parallel processing elements in a system.

Parallelism from such independent tasks can be easily exploited via work queues, where each task in the queue can be assigned to an available compute element. Note that any of these three partitions of the graph

can be an empty set; if empty set production is the frequent case, the number of independent tasks may grow more slowly than expected.

The key observation behind the Trim [51] step is that a trivial SCC (i.e. SCC of size one) is easy to identify: it has either zero incoming edges or zero outgoing edges in the current partition. Therefore, one can easily identify such trivial SCCs only by looking at the number of neighbors, rather than by computing two reachable sets, which is computationally more expensive.

The Trim step can be repeated iteratively, since trimming a node can cause other nodes to become trivial SCCs. Figure 4.1(b) illustrates this idea. In the figure, nodes c, d, and e can be identified as trivial SCCs quickly, as they have zero in- or out-degree and thus cannot form a cycle. The trimming of node c in turn makes node b a trivial SCC, whose trimming also makes node a trivial.

Algorithm 1: FW-BW-Trim(G, SCC)

In-Out: G: a graph (a subgraph of the original input graph)
In-Out: SCC: a collection of node sets; each set corresponds to an SCC of the original graph
Trim(G, SCC)
if $|Nodes(G)| = 0$ **then return**;

$u \leftarrow$ pick any node in G /* pivot */
$FW \leftarrow$ Forward-Reach(G, u)
$BW \leftarrow$ Backward-Reach(G, u)
$S \leftarrow FW \cap BW$
$SCC \leftarrow SCC \cup \{S\}$
begin in parallel
  FW-BW-Trim($FW \setminus S$, SCC)
  FW-BW-Trim($BW \setminus S$, SCC)
  FW-BW-Trim($G \setminus (FW \cup BW)$, SCC)

Algorithm 2: Trim(G, SCC)

In-Out: G: a graph (a subgraph of the original input graph)
In-Out: SCC: a collection of node sets; each set corresponds to an SCC of the original graph
repeat
  foreach $n \in G$ **do**
    if *In-degree*$_G(n)$ *= 0* $\vee$ *Out-degree*$_G(n)$ *= 0* **then**
      $SCC \leftarrow SCC \cup \{\{n\}\}$
      $G \leftarrow G \setminus \{n\}$
until G *not changed*

The FW-BW-Trim algorithm is described in Algorithm 1; Algorithm 2 shows details of the Trim step. Although Trim is a simple idea, it greatly improves the performance of the previous FW-BW algorithm,

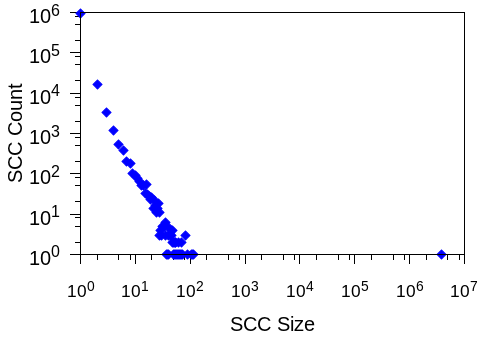


Figure 4.2: Distribution of SCC sizes in the LiveJournal network.

especially for *real-world* graphs [11]. Therefore, to understand its effectiveness, one must comprehend the characteristics of *real-world* graphs.

4.4 Our Parallel SCC Algorithm

In this section, we discuss our extensions to the conventional FW-BW-Trim algorithm, which account for the characteristics of *real-world* graphs.

Algorithm 3: Baseline(G, SCC)

Input : G, the original input graph
Output: SCC, a collection of node sets
Local : $Color$, color value assigned to each node in G
Local : $mark$, boolean value assigned to each node in G
/* Initialization */
$\forall\, n \in G$: $Color(n) \leftarrow 0, mark(n) \leftarrow false$

Par-Trim(G, SCC, $Color$, $mark$)
Recur-FWBW(G, 0, SCC, $Color$, $mark$)
until *work queue is empty* **do in parallel**
  $c \leftarrow$ pop a color from the work queue
  Recur-FWBW(G, c, SCC, $Color$, $mark$)

4.4.1 Baseline Implementation using Parallel Trim

We prepare an efficient implementation of the conventional FW-BW-Trim algorithm and set it as our baseline (Algorithm 3); it has a few small enhancements over Algorithm 1. These improvements include parallelization of the Trim step, the use of additional data structures to avoid directly modifying the input graph, and a work queue to support parallelism in recursion, described below.

Algorithm 4: Par-Trim(G, SCC, $Color$, $mark$)

```
Input  : G, the original input graph
In-Out: SCC, a collection of node sets
In-Out: Color, color value assigned to each node in G
In-Out: mark, boolean value assigned to each node in G
repeat
    foreach n ∈ G, mark(n) = false do in parallel
        /* only count neighbors with same color as n                    */
        if In-degree(n,Color) = 0 ∨
        Out-degree(n,Color) = 0 then
            Color(n) ← -1
            SCC ← SCC ∪ {{n}} ; mark(n) ← true
until Color not changed
```

The Baseline algorithm (Algorithm 3) has two phases: first, it performs the Trim operation in parallel on multiple disconnected nodes, shown in Algorithm 4, and second, it applies the conventional recursive FW-BW algorithm, shown in Algorithm 5, using a work queue. Since there are many size-1 SCCs in a *real-world* graph, the parallel trim step greatly increases the degree of parallelism by identifying these SCCs before executing the FW-BW algorithm. Although Par-Trim is invoked once at the beginning of Algorithm 3, the actual trimming is iteratively applied inside the Par-Trim kernel. The example in Figure 4.1(b) demonstrates this idea: the trimming of nodes c, d, and e can be completed in parallel, followed by iterative trimming of nodes b and a.

For performance reasons, we do not mutate the input graphs directly. Instead, we use two auxiliary data structures: *mark* and *Color*. When the SCC of a node is identified, instead of detaching the node from the rest of the graph, we simply set the *mark* value of the node to *true*, and the node is considered detached thereafter. Similarly, when we partition the graph, we assign the same *Color* value to nodes belonging to the same partition; each partition is assigned a unique *Color* value. Therefore, two nodes of different *Color* values are considered disconnected, even when there exists an edge between them in the original graph. Algorithm 4 and Algorithm 5 show how these data structures are used, and their implementation is discussed in Section 4.5.

Recursion parallelism for the Recur-FW-BW kernel is implemented through a work queue. At the end of Algorithm 5, the three remaining partitions (c, c_{fw}, c_{bw}), other than the newly identified SCC, are pushed into the shared work queue. Every worker thread in the system grabs a partition (i.e. *Color*) from the work queue and processes it concurrently with respect to other worker threads. The program is finished when all the workers become idle and no work items remain in the queue.

Algorithm 5: Recur-FWBW(G, c, SCC, $Color$, $mark$)

Input : G, the original input graph
Input : c, a color value
In-Out: SCC, a collection of node sets
In-Out: $Color$, color value assigned to each node in G
In-Out: $mark$, boolean value assigned to each node in G
$pivot \leftarrow$ choose a random node $s.t.\ Color(pivot) = c$
if $pivot$ = *NIL* **then return**

$c_{fw}, c_{bw}, c_{scc} \leftarrow$ a new color value (each)
$S \leftarrow \emptyset$
traverse G from $pivot$ using forward edges
  when visiting *node* n **do**
    if $Color(n) = c$ **then**
      $Color(n) \leftarrow c_{fw}$
    else prune traversal beyond n;
traverse G from $pivot$ using reverse edges
  when visiting *node* n **do**
    if $Color(n) = c$ **then**
      $Color(n) \leftarrow c_{bw}$
    else if $Color(n) = c_{fw}$ **then**
      $S \leftarrow S \cup \{n\}$; $mark(n) \leftarrow True$
      $Color(n) \leftarrow c_{scc}$
    else prune traversal beyond n;
$SCC \leftarrow SCC \cup \{S\}$
push c, c_{fw}, c_{bw} into the work queue

4.4.2 Method 1: Two-Phase Parallelization

Section **??** introduced two important properties of SCC structures in *real-world* graphs: (1) there exists a giant SCC whose size is O (N), and (2) there are many small sized SCCs, where the number of SCCs of a given size decreases drastically as the size grows. Moreover, studies of the SCC structure in *small-world* graphs also revealed that the giant SCC can be considered the *center*, to which most of the other small SCCs are attached [19, 41].

What is the implication of this SCC structure to the performance of the conventional FW-BW-Trim algorithm? Most of all, it causes workload imbalance in the algorithm. The conventional implementation of the FW-BW-Trim algorithm lets each thread find one SCC at a time, though there exists one O (N)-sized giant SCC in the graph. Worse, it is very likely that this giant SCC is identified at the beginning because other

small SCCs are weakly connected to this giant SCC. Consequently, while the large SCC is being identified by one thread, all the other threads stay idle since there are no other tasks.

Based on the above observations, we adopt another two-phase parallelization strategy. In phase 1, we exploit data-level parallelism, letting every thread work on the same partition of the graph; all threads are used to find reachable sets. In phase 2, we return to the conventional implementation, which exploits task-level parallelism. The transition between phase 1 and phase 2 occurs when the giant SCC has been identified (i.e. an SCC containing, say 1% of the nodes of the original graph), or after a predefined number of iterations.

This strategy is summarized as Method 1 in Algorithm 6. We omit the detailed description of Par-FWBW since it is almost identical to Algorithm 5 except that the traversal of the graph is implemented with parallel breadth-first search, and the parallel BFS is repeated until the giant SCC (e.g. an SCC containing more than 1% of nodes) is identified or given the maximum number of trials. Note that a BFS on *small-world* graphs results in a small number of BFS levels, but a large number of nodes in each level that can be visited in parallel [35]. Also, the algorithm applies parallel Trim once more after the Par-FWBW step because detection of the giant SCC may present an opportunity for further trimming.

Algorithm 6: Method1(G, SCC)

Input : G, the original input graph
Output: SCC, a collection of node sets
Local : $Color$, color value assigned to each node in G
Local : $mark$, boolean value assigned to each node in G

/* Initialization */
$\forall\, n \in G$: $Color(n) \leftarrow 0, mark(n) \leftarrow false$

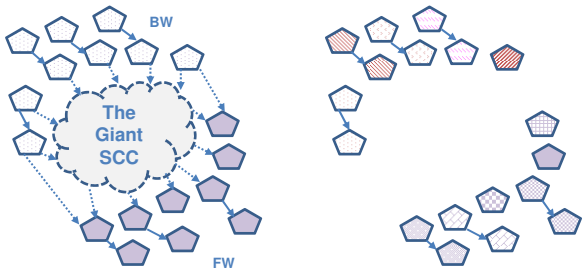
/* Phase 1: parallelism in trims and traversals */
Par-Trim(G, SCC, $Color$,$mark$)
Par-FWBW(G, 0, SCC, $Color$, $mark$)
Par-Trim(G, SCC, $Color$,$mark$)

/* Phase 2: parallelism in recursion */
until *work queue is empty* **do in parallel**
  $c \leftarrow$ pop a color from the work queue
  Recur-FWBW(G, c, SCC, $Color$, $mark$)

4.4.3 Finding Weakly Connected Components

Method 1 in the previous subsection successfully parallelizes detection of SCCs for most *real-world* graph instances, as shown in the experiments (Section 4.6). This occurs because most of the nodes in *real-world* graphs are processed in a data parallel phase of the algorithm.

However, the second phase of the algorithm, the recursive FW-BW step, is scarcely parallelized even when a large number of SCCs (e.g. 100,000) are identified in this phase. In fact, especially when a large



(a) (b)

Figure 4.3: SCC structure of small-world graphs: (a) when the giant SCC is identified and removed, and (b) after the weakly connected component detection algorithm has been applied. Each polygon represents a small-sized SCC. In (a), same color polygons belong to the same set.

proportion of nodes are processed in the second phase, such limited parallelism diminishes the overall parallel speedup of Method 1.

The first clue to explain this phenomenon was found in the work queue logs; the recorded maximum queue depth with single threaded execution is only six, indicating insufficient task-level parallelism. This was counter-intuitive at first, because the FW-BW algorithm is designed to produce three more tasks for each task being processed. To understand why, again we must consider the shape of *small-world* graphs.

Figure 4.3(a) illustrates a typical SCC structure of small-world graphs according to previous studies [19, 41], where the small SCCs are connected around the giant SCC. Now consider the moment when the giant SCC has been identified by the FW-BW algorithm. Ignoring non-connected SCCs for the time being, the remaining SCCs are grouped into two sets (colors): the FW-set and the BW-set. However, many of these SCCs are not connected to each other. Therefore, recursive application of the FW-BW algorithm to each set (color) will only identify one SCC to which the pivot belongs, but does not provide further partitioning. Consequently, the execution is serialized.

Following is the log of the first five task executions in the recursive FW-BW step when Method 1 is applied to a large graph instance named Flickr (Section 4.6). The SCC column indicates the size of the SCC identified in the iteration, and FW, BW, and Remain indicate the resulting forward, backward, and remaining set sizes, respectively. The log verifies that our observation above indeed occurs in Method 1; each task execution identifies only a small SCC and fails to create additional tasks (i.e. FW and BW sets).

```
SCC FW BW Remain
2   0 0 125432
5   0 0 125427
11  0 0 125416
3   3 0 125410
...
```

The above observation, however, also suggests a way to solve this problem. Once the giant SCC has been

identified, the remaining graph is composed of many small components that are disconnected from each other. Groups of one or more of these disconnected SCCs form weakly connected components (WCCs), where a WCC is defined as a maximal group of nodes that are mutually reachable by converting directed edges to undirected edges. Therefore, we identify all of the weakly connected components over the whole graph in parallel, and assign each WCC a different color. Then, each WCC becomes a separate entry in the work queue, resulting in a substantial improvement in the degree of task-level parallelism in the recursive FW-BW phase. Figure 4.3(b) illustrates this idea.

Algorithm 7: Par-WCC(G, $Color$, $mark$)

```
Input  : G, the original input graph
In-Out: Color, color value assigned to each node in G
In-Out: mark, boolean value assigned to each node in G
Local  : WCC, head-node value assigned to each node in G
forall the n ∈ G, mark(n) = false do in parallel
    WCC(n) ← n
repeat
    foreach n ∈ G, mark(n) = false do in parallel
        foreach k ∈ OutNbr(n), Color(k) = Color(n) do
            if WCC(k) < WCC(n) then
                WCC(n) ← WCC(k)
    foreach n ∈ G, mark(n) = false do in parallel
        k ← WCC(n)
        if k ≠ n ∧ k ≠ WCC(k) then
            WCC(n) ← WCC(k)
until WCC not changed
foreach n ∈ G, WCC(n) = n do in parallel
    c ← a new color
    foreach k ∈ G, WCC(k) = n do
        Color(k) ← c
    push color c into the work queue
```

Algorithm 7 details how to find weakly connected components in parallel. Once the WCCs are identified, they are pushed into the work queue. We use *Color* and *mark* in the same way as in Algorithm 4, i.e. the Par-WCC algorithm assigns a node's out-neighbors to its WCC only when they are the same color.

4.4.4 Trim2: Fast Detection of Size-2 SCCs

We also add a fast parallel detection mechanism for size-2 SCCs, namely Trim2. The idea is that a large subset of size-2 SCCs can be detected easily by looking only at the neighbors of a given node. Figure 4.4

Algorithm 8: Par-Trim2(G, SCC, $Color$, $mark$)

Input : G, the original input graph
In-Out: SCC, a collection of node sets
In-Out: $Color$, color value assigned to each node in G
In-Out: $mark$, boolean value assigned to each node in G
foreach $n \in G$, $mark(n) = false$ **do** in parallel
  if *In-degree(n,Color) = 1* **then**
    $k \leftarrow$ the only *InNbr* of n
    if $k \in OutNbr(n) \wedge$ *In-degree(k,Color) = 1* **then**
      $Color(n), Color(k) \leftarrow$ -1
      $mark(n), mark(k) \leftarrow true$
      $SCC \leftarrow SCC \cup \{\{n, k\}\}$
  else if *Out-degree(n,Color) = 1* **then**
    $k \leftarrow$ the only *OutNbr* of n
    if $k \in InNbr(n) \wedge$ *Out-degree(k,Color) = 1* **then**
      $Color(n), Color(k) \leftarrow$ -1
      $mark(n), mark(k) \leftarrow true$
      $SCC \leftarrow SCC \cup \{\{n, k\}\}$

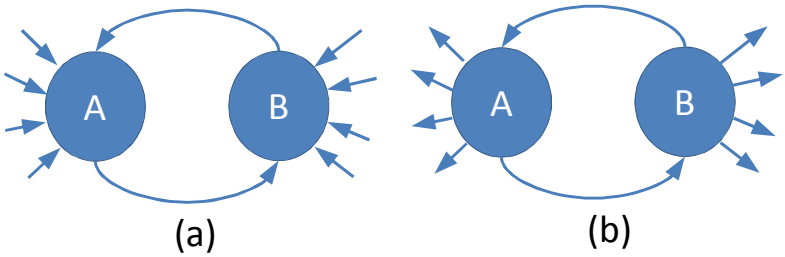


Figure 4.4: Patterns of size-2 SCCs detected by Trim2. There is a tight cycle between A and B but either (a) there are no other outgoing edges from A and B, or (b) there are no other incoming edges to A and B.

illustrates the patterns of size-2 SCCs identified by this algorithm. The algorithm first identifies all of the nodes which have a single neighborhood node that is both an incoming neighbor and an outgoing neighbor, i.e. nodes A and B in Figure 4.4. Then the algorithm examines the original node's sole neighbor. If the neighbor has no incoming (or outgoing) edges other than to the original node, the algorithm identifies these two nodes as an SCC because there cannot be any larger cycle that contains both nodes.

The detailed Trim2 algorithm is summarized in Algorithm 8. Unlike Trim, which is applied multiple times iteratively, we apply Trim2 only once since it is computationally more expensive. Our experiments revealed that the Trim2 step provides only a marginal speedup by itself; however it reduces the execution time of the following WCC step by up to 50% because it cuts out a chain of weakly connected size-2 SCCs. For this reason, we include Trim2 only for Method 2, described in Section 4.4.5.

Algorithm 9: Method2(G, $Color$)

Input : G, the original input graph
Output: SCC, a collection of node sets
Local : $Color$, color value assigned to each node in G
Local : $mark$, boolean value assigned to each node in G

/* Initialization */
$\forall\, n \in G$: $Color(n) \leftarrow 0, mark(n) \leftarrow false$

/* Phase 1: parallelism in trims, traversals and WCC */
Par-Trim(G, SCC, $Color$, $mark$)
Par-FWBW(G, 0, SCC, $Color$, $mark$)
Par-Trim′(G, SCC, $Color$, $mark$)
Par-WCC(G, $Color$,$mark$)

/* Phase 2: parallelism in recursion */
until *work queue is empty* **do in parallel**
  $c \leftarrow$ pop a color from the work queue
  Recur-FWBW(G, c, SCC, $Color$, $mark$)

4.4.5 Method 2: Putting It Together

Our Method 2, summarized in Algorithm 9, includes all of the above steps applied in sequence. Here, Par-Trim′ includes the application of Par-Trim (iteratively), Par-Trim2 (only once), and Par-Trim (iteratively). We only apply Par-Trim2 once because it is computationally more expensive than Par-Trim. The primary difference between Method 1 and Method 2 is the inclusion of the Par-Trim2 and Par-WCC steps. The performance differences between the two methods are discussed in Section 4.6.

4.5 Issues in Implementation

We implement efficiently in C++ our two methods and the Baseline algorithm from Section **??** as well as Tarjan's algorithm. There are several pitfalls in implementing these algorithms and a careless implementation could result in an order of magnitude lower performance.

4.5.1 Graph and Set Representation

We implemented all of the algorithms in the paper using C++. For the in-memory graph data structure, we used the compressed sparse row (CSR) format, which uses two arrays to represent the graph. A $\mathrm{O}(N)$-sized array stores a pointer to the beginning of each node's adjacency list, stored in a single $\mathrm{O}(M)$-sized array (see Figure 4.5). Note that CSR is favored in high performance graph analysis problems [8, 35, 11] because it is compact, memory bandwidth-friendly, and thus best suited for graph traversals.

The CSR representation, however, performs poorly when modifying the graph structure itself. Therefore, instead of actually removing nodes that are trimmed or whose SCCs are identified, we maintain the extra data structures $mark$ and $Color$.

$mark$ is a $\mathrm{O}(N)$ boolean array which represents the nodes whose SCCs are identified and thus are detached from the original graph. Our algorithm always ignores nodes whose $mark$ value is $true$; therefore, setting the $mark$ value of a node has the same effect as removing the node from the graph representation.

Similarly, $Color$ is a $\mathrm{O}(N)$ integer array which represents the current partitioning of the graph. That is, all the nodes in a (non-trivial) partition of the graph have the same *color* value, and a unique *color* value is assigned to each partition. For instance, the construction of the FW-set and BW-set in Algorithm 5 simply assigns the same *color* value to every reachable node.

Therefore, neighborhood nodes whose color is different from the current node are considered detached.

However, this approach presents an issue when selecting a pivot from a specific set (Algorithm 5). To select any single node of a specific *color*, the complete *Color* array must be scanned; this becomes a very expensive operation when there exist only a few nodes of that *color*.

To solve this issue, we adopt a hybrid representation. That is, while constructing FW-set and BW-set in Algorithm 5, we maintain a compact representation (i.e. `std::set`) for each set, in addition to the *Color* array. The former is used to choose pivots in the FW-BW step, while the latter is used to look up membership of a set. However, we use the hybrid representation only for phase 2 (i.e. Recur-FWBW) but not for phase 1, since the set size of each color is sufficiently large in phase 1. Our experiments revealed that such a hybrid approach resulted in $\sim$10x better performance than using one representation only.

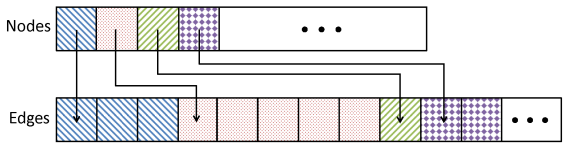


Figure 4.5: CSR adjacency matrix data structure: the node array stores pointers to each node's adjacency list in the edge array.

4.5.2 Implementing Graph Traversals

The classic Tarjan's SCC algorithm is based on a depth-first search (DFS) traversal of the graph. However, the required recursion depth for DFS traversal is the size of the largest SCC, which is $\mathrm{O}(N)$ for large real-world graphs. Thus, one must increase the size of the program stack accordingly, to hundreds of MBs or even a few GBs. Moreover, Tarjan's algorithm requires an additional stack (other than the program stack) on which it places nodes in the order in which they are visited; the algorithm must check if a node is in this stack. Like the $Color$ array and `std::set` representations described in **??**, we implement this stack using both a vector and a boolean array for fast execution.

For the reachable set computation in the parallel FW-BW step, we used an efficient implementation of the breadth-first search (BFS) order graph traversal [35, 13]. Note that after the advent of the graph500 benchmark suite [2], many efficient implementations of the BFS traversal have been proposed [52, 66], which may improve our performance results even further.

On the other hand, for the same computation in the recursive FW-BW step, we use DFS instead of BFS. This is because the BFS implementation above, optimized for parallel traversal, has a larger fixed cost than simple sequential DFS. Also, during reachable set exploration in the parallel FW-BW step, we do not maintain an unbounded set representation (i.e. `std::set`), but use the *Color* array only. This is based on the following observations: (1) the traversal will go through a huge fraction of nodes in the graph (i.e. $\mathrm{O}(N)$) and thus the size of each set (FW-set, BW-set, and remaining set) will be large as well, and (2) those sets will be modified by the following trimming and compacting operations. Therefore, we defer the construction of sets until the end of the trimming phase, when we perform a scan of non-marked nodes to construct the initial work items.

4.5.3 Managing Parallel Work Items

For the threading library, we used OpenMP for all experiments. As a reminder, we exploited data-level parallelism in the first phase of our algorithms, but task-level parallelism in the second phase. The data-level parallelism is implemented using the `parallel for` statement, and the task-level parallelism with a custom work queue implementation. For the data-level parallelism, however, it was critical to specially handle the workload imbalance problem. Note that there is another fundamental characteristic of *real-world*

Name	Description	# Nodes	# Edges	Largest SCC Size	Diam
Livej	Links in LiveJournal (Web) [7],[46]	4,848,571	68,993,773	3,828,682	
Flickr	Connection of Flickr users (Social) [53]	2,302,925	33,140,018	1,605,184	
Baidu	Links in Baidu Chinese online encyclopedia (Web) [55]	2,141,300	17,794,839	609,905	
Wiki	Links in English Wikipedia (Web) [6]	15,172,740	131,166,252	4,736,008	
Friend*	Connection of Friendster users (Social) [76]	124,836,180	1,806,067,135	46,941,703	
Twitter	Connection of Twitter users (Social) [42]	41,652,230	1,468,365,182	33,479,734	
Orkut*	Connection of Orkut users (Social) [76]	3,072,627	117,185,083	2,963,298	
Patents	Citation among US Patents [45]	3,774,768	16,518,948	1	
CA-road*	Road network of California [46]	1,965,206	5,533,214	1,168,580	

Table 4.1: Real-world graph datasets used in the experiments. * indicates that the original graph is undirected; we randomly assign a direction for each edge with 50% probability for each direction. The graph diameters are estimated from a random sampling of nodes; the actual diameters are likely somewhat larger due to outlier nodes.

graphs, the scale-free property, which means that the graph's degree distribution follows a power law [10]. In other words, there exist a few nodes that have a huge number of neighbors while many nodes have only a few neighbors. Therefore, statically assigning the same number of nodes to each thread naturally induces workload imbalance if the work involves neighborhood exploration. Thus, we used dynamic load-balancing for the components that involve neighborhood exploration, but static workload distribution otherwise.

For the task-level parallelism, we used our custom work queue implementation, which is composed of two levels of queues: a global queue and per-thread private queues. Initially, each thread fetches up to K work items from the global queue into its local queue; whenever the local queue becomes empty, more work is fetched from the global queue. Each newly generated work item goes to a local queue first. When the size of a local queue grows to $2K$, K items are moved to the global queue. We set K to 1 for the Baseline and Method 1, because these algorithms suffer from a lack of task level parallelism; for Method 2, we set K to 8.

4.6 Experiments

In this section, we evaluate the performance of our methods on several large *real-world* graph instances that are available from public repositories [47, 3]. We have chosen graph instances that are large enough to parallelize (i.e. more than 10 million edges). Table 4.1 summarizes the size of each graph and provides a short description of the graph instance.

All of our experiments were performed on a commodity server-class machine with two Intel Xeon E5-2660 (2.20GHz) CPUs, each of which has 8 cores and 16 hardware threads. There are in total 20 MB of last-level cache and 256 GB of main memory. For all implementations, we used OpenMP for the threading library and compiled our code with g++ version 4.4.7 with the -O3 option. Finally, our servers are running

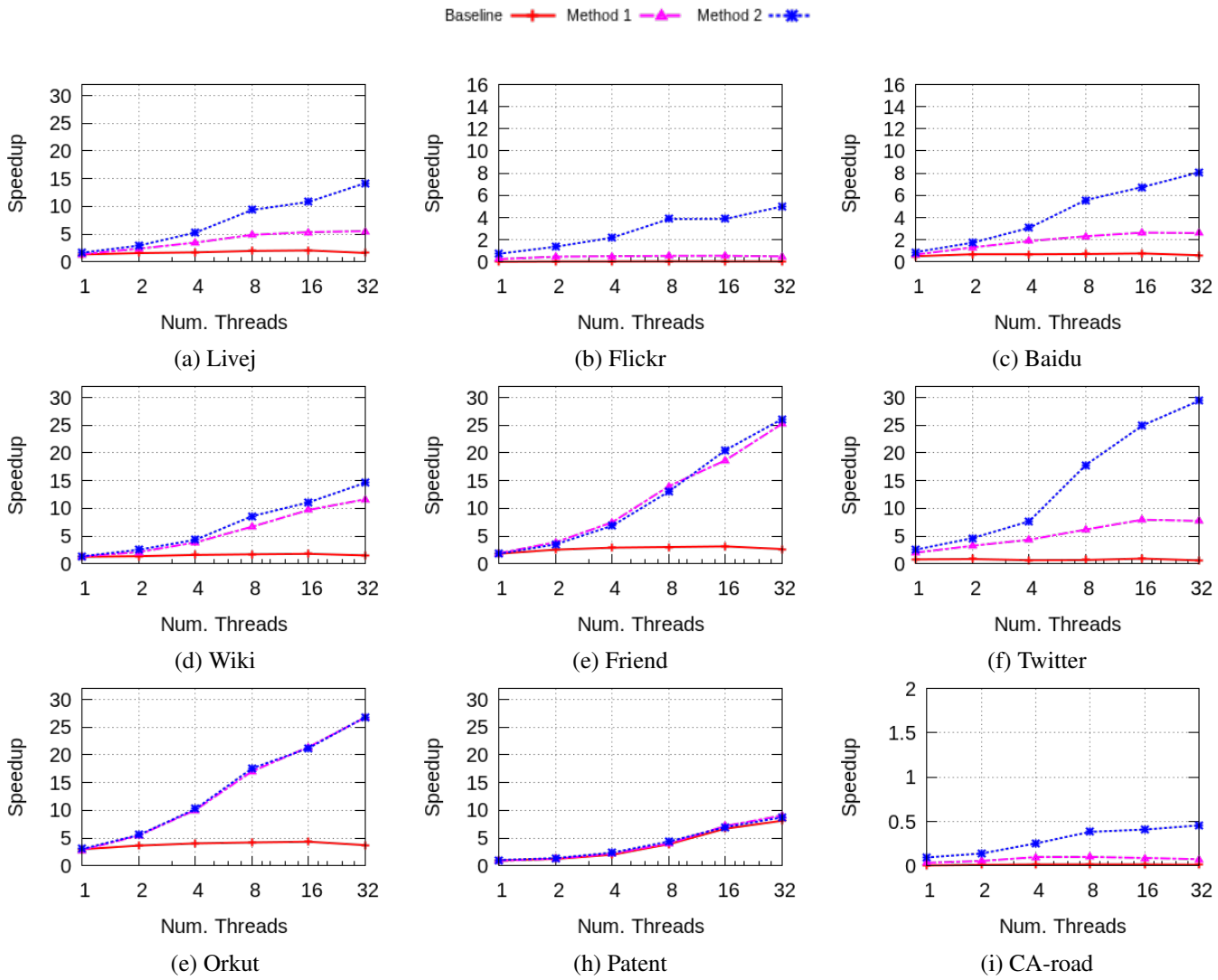


Figure 4.6: Performance results on real-world graph instances. The y-axis is speedup compared to the optimal sequential algorithm (i.e. Tarjan's). The x-axis is in log scale. Note that each of two CPU sockets has only 8 cores; 16-thread execution exploits two sockets and 32-thread execution uses simultaneous multithreading. The Baseline (Algorithm 3) uses parallel trim and the recursive FW-BW algorithm; Method 1 (Algorithm 6) utilizes two-phase parallelization (data-level and task-level); Method 2 (Algorithm 9) adds parallel trim2 and parallel WCC.

the CentOS Linux (6.4 Final) operating system.

The plots in Figure 4.6 summarize the performance of our methods on the real-world graph instances in Table 4.1. The y-axis is the speedup against Tarjan's optimal sequential algorithm, and the log scale x-axis is the number of threads.

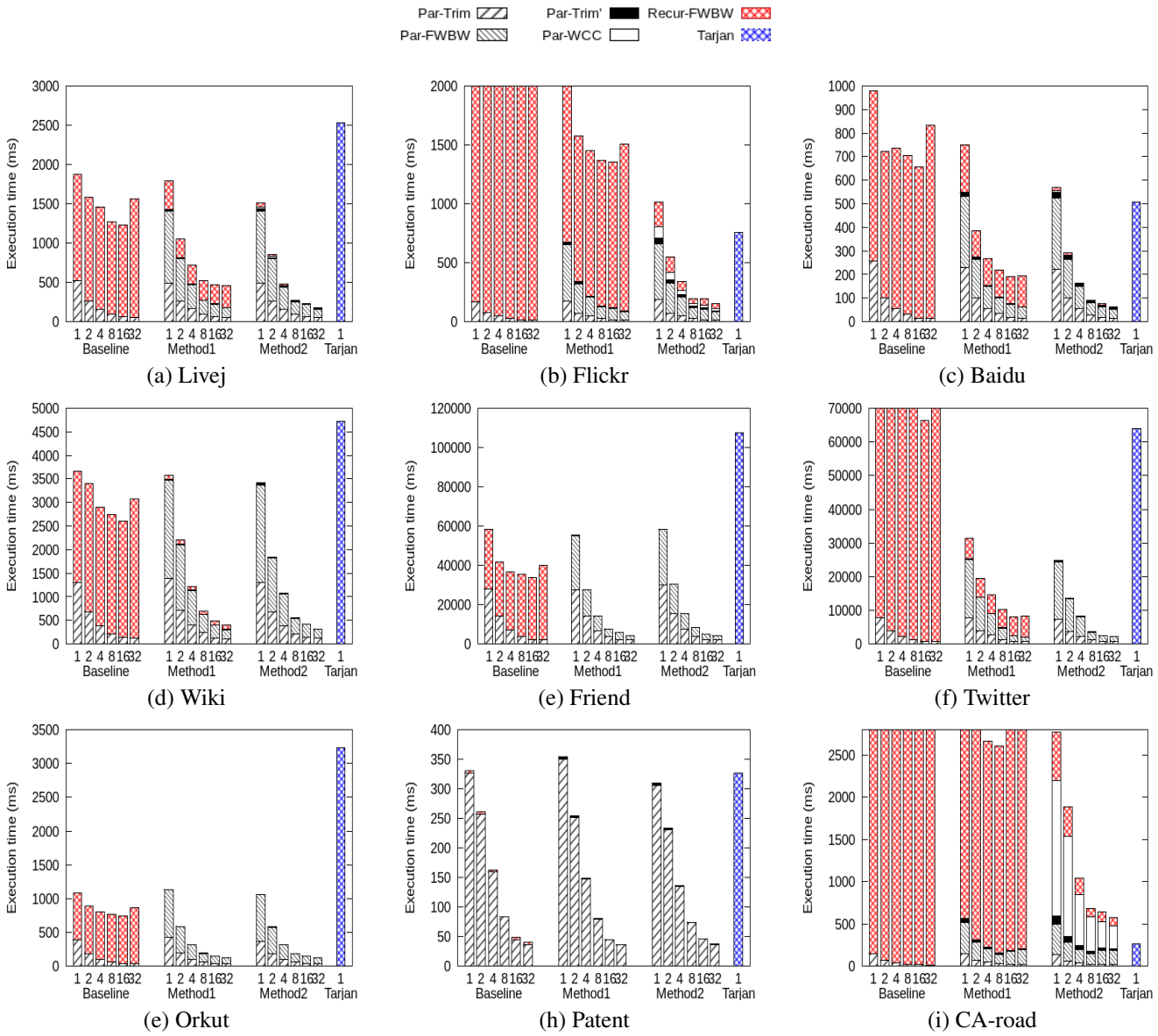


Figure 4.7: Execution time breakdown for all methods on all graph instances. Par-Trim′ accounts for applying Trim only for Method 1 but applying Trim, Trim2 and Trim in sequence for Method 2.

A first look over all instances (except CA-road, which we will discuss later) reveals that our methods not only improve the performance of the baseline implementation of the FW-BW-Trim algorithm, but also exploit a greater degree of parallelism. Excluding CA-road, the speedup result varies from 5.01x (Flickr) to 29.41x (Twitter) using 16 cores and 32 hardware threads. The geometric mean speedup is 14.05x. Also, we

can see that Method 2 provides further performance improvement over Method 1 for certain graph instances.

We remind the reader that the machine has two CPU sockets, where each CPU has 8 cores only. As a result, there is a natural knee in Figure 4.6 between 8 threads and 16 threads, since the latter crosses the socket boundary, i.e. NUMA effect. Similarly, there is another knee between 16 threads and 32 threads, because the latter exploits simultaneous multithreading (SMT) using two hardware threads in each physical core.

To better understand the performance behavior shown in Figure 4.6, we plot in Figure 4.7 the execution time breakdown of each method for all of the graph instances. The y-axis in the plots is the execution time measured in milliseconds. Thus, each vertical bar segment represents the time spent in each phase of the algorithm.

Figures 4.6 and 4.7 first show that the Baseline method does not scale. As explained in Section 4.4, a singe thread processes the gigantic SCC in each graph, thus the recursive FW-BW phase (the topmost segment) rarely exploits parallelism. To the contrary, the parallel FW-BW phase of Method 1 (Section 4.4.2) detects the largest SCC of the graph in parallel, which is essential to achieve overall speedup. You can see this in Figure 4.7, where the second to bottom segments (Par-FWBW) scale down as we increase the number of threads, representing a diminishing fraction of the total execution time. Consequently, Method 1 provides a fair amount of parallel speedup as shown in Figure 4.6.

Next we look at the cases where Method 2 provides an additional performance benefit over Method 1, including Livej, Flickr, Baidu, and Twitter. Notice that in Figure 4.7(b), the execution time of the recursive FW-BW phase (the topmost segment) for Method 1 does not scale down even with more threads. The reason for this phenomenon has been explained in Section 4.4.3: each step in the recursive FW-BW phase does not partition the remaining graph well, failing to provide sufficient parallelism.

Figures 4.6 and 4.7 also confirm that Method 2 successfully solves this issue. As can be seen in Figure 4.7(b), the execution time of the recursive FW-BW phase now scales down in Method 2, due to introduction of the parallel WCC phase. Our execution log also confirms that at the beginning of the recursive FW-BW phase there are about 10,000 work items in the queue, providing sufficient task-level parallelism. Moreover, the parallel WCC phase itself is well parallelized, as its execution time decreases with increasing number of threads. Therefore, the actual benefits of Method 2 over Method 1 depend on the structure of the graph instance. To illustrate this point, Figure 4.8 shows the fraction of nodes whose SCCs are identified by each phase. Noticeably, the more nodes identified by the recursive FW-BW step, the more performance benefits are achieved by Method 2.

Finally, we discuss the case of the CA-road graph. The graph does not share the same characteristics as the other graph instances because it is (almost) planar by its nature. Therefore, the assumptions that we have made in Section 4.4 do not stand for this non-small-world graph instance. First, the graph has a large diameter (~ 1000) and thus does not possess the small-world property. Second, even though the graph still has a giant SCC, it also has many more large-sized SCCs than small-world graphs (see Figure 4.9).

Figure 4.9 shows the SCC structure of all graphs used in the experiments discussed in this section. Notice that there is a single giant connected component whose size is $O(N)$, the most frequent SCCs are size one, and there are SCCs of other sizes in between for all graph instances except Patent. These in-between-sized SCCs differentiated scalability between Method1 and Method2 (Figure 4.6), as described in Sections 4.4.3 and 4.4.4.

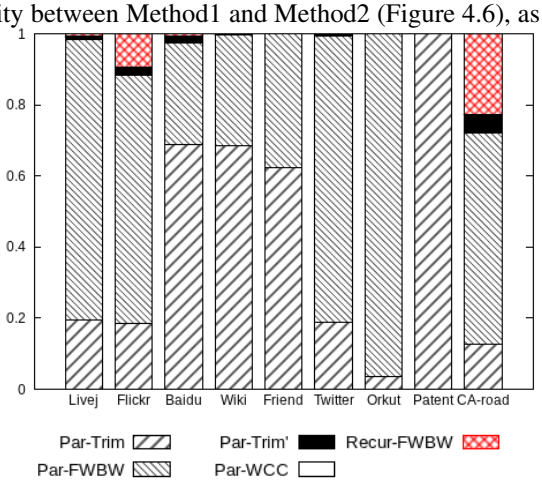


Figure 4.8: Fraction of nodes whose SCC is identified at each phase of execution for Method 2.

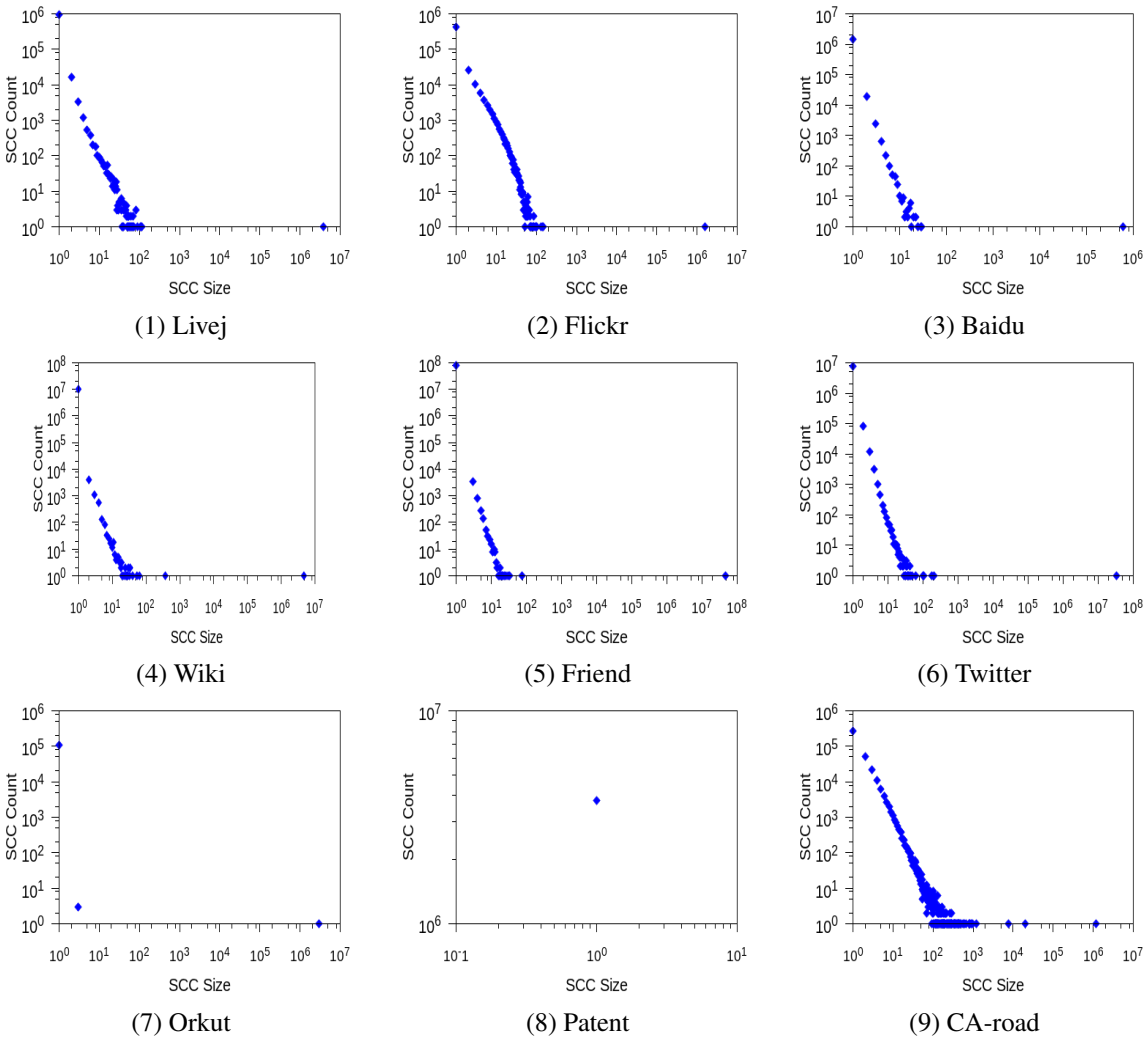
Patent is a special case with no cycles in the graph. However, this is a natural phenomenon due to the way the graph is constructed: a patent can only cite other patents that come before it, thus preventing any cycles. Note that the SCC structure of this graph was identified by the Trim operation.

CA-road also shows a noticeably different distribution, since it is not a *small-world* graph. Having a large diameter, the graph has many more non-trivial SCCs than the other graphs. Moreover, the size of these SCCs is larger as well.

As a result, the parallel FW-BW step provides rather limited parallel speedup in this case because the level-synchronous BFS does not scale up well in such graphs [35]. Moreover, the performance of Method 2 decreases as the execution time of the WCC algorithm increases; the algorithm requires a large number of iterations for convergence when applied on *non-small-world* graphs.

Thus, both methods, although they still scale, do not perform as well as Tarjan's method for CA-road. Nevertheless, we remind the reader that in the common case, users have a priori knowledge about the property of their graphs, small-world or not. Also, small-world graphs draw more research interest because they are the dominant class of natural large graph instances for many important applications where the graphs are constructed by arbitrary relationships. For example, all of the large graph instances other than the road networks in public repositories [47, 3] have the *small-world* property.

In summary, our experiments validate the success of our methods in parallelizing SCC detection algorithms for *small-world* graphs because our methods effectively exploit fundamental characteristics of those graphs.



(1) Livej (2) Flickr (3) Baidu

(4) Wiki (5) Friend (6) Twitter

(7) Orkut (8) Patent (9) CA-road

Figure 4.9: Distribution of SCC sizes of the graph instances that are used in our experiments.

4.7 Conclusion

In this paper, we analyze the performance shortcomings of the conventional FW-BW-Trim algorithm when applied to *small-world* graph instances. We propose three simple extensions to the conventional algorithm that take advantage of *small-world* graph properties to address the deficiencies in existing algorithms. Consequently, our extensions result in significant parallel and sequential performance improvements on *small-world* graph instances to deliver state-of-the-art parallel SCC detection performance.

As a next step, we plan to implement our algorithm in a distributed environment. Our extensions can be

easily implemented in such an environment as they only require data from direct neighbors.

Chapter 5

Cache Optimizations to Improve Performance

5.1 Introduction

The data-intensive nature of graph algorithms (discussed in Chapter 3) means that optimizations that decrease the total memory bandwidth requirement and/or decrease average data access latency will positively impact overall performance. The primary mechanisms used for this in modern multicore CPUs are prefetching and caches.

When a piece of data misses in the cache, there are four possible causes of the miss. 1) Compulsory or cold: data has not been previous accessed. 2) Capacity: data was previously accessed and cached, but has since been evicted due to insufficient space in the cache. 3) Conflict: data was previously accessed and cache, but since evicted due to access of other data which maps to the same set. 4) Coherence: data was previously accessed and cached, but then written by another processor, making the accessing processor's copy invalid.

TODO: how to distinguish between capactiy and conflict misses?

Conflict misses are easily addressed by increasing the associativity of the cache. Modern caches often have 8- or 16-way associativity, and conflict misses represent <1% of total cache misses for the graph algorithms we study (TODO: measure this).

Prefetching can address both compulsory and capacity misses, but is the only way to affect compulsory misses.

The cache replacement policy can affect capacity misses, by making a better selection of which data to evict from the cache. An optimal cache replacement policy will evict the piece of data in the target set which

will be accessed the farthest in the future, but this can only be determined by looking into the future.

The common least-recently used (LRU) replacement policy is vulnerable to thrashing, or repeated swapping in and out of cache data, which can occur for memory-intensive workloads with a working set larger than the cache size, as is the case for graph algorithms.

[TODO: explain thrashing, maybe put this in characterization section?]

We leverage information about the memory access patterns in graph algorithms, described in Section **??** to create a combined software-hardware approach to the cache replacement policy.

We focus on the cache replacement policy as a mechanism to improve the cache hit rate, and consequently, the performance, of graph algorithms.

5.2 Related Work

[TODO: How to narrow down space of related work?] single application in multi-threaded context, single-threaded applications

The prototypical cache replacement policy is least-recently used (LRU) replacement, which has been the primary policy in modern processors until the last few years, when more sophisticated policies have been implemented in Intel LLCs, for example. [TODO: cite] LRU is typically implemented as a similarly performing and more space efficient pseudo-LRU policy. [TODO: cite]

We focus our study of related work on CPU cache replacement policies, both those implemented exclusively in hardware and with a combined software/compiler and hardware approach.

There are two important contributions in the hardware-only domain. Qureshi et al. [60] present their Dynamic Insertion Policy (DIP), which chooses between two insertion policies using set dueling. Set dueling dedicates a few sets of the cache to each of the two policies. The remaining follower sets use the policy that has fewer misses on the dedicated sets. The LRU Insertion Policy (LIP) places the incoming line in the LRU position, rather than the MRU position. The Bimodal Insertion Policy (BIP) is an enhancement to LIP that adapts to working set changes. DIP reduces the average MPKI by 21% for SPEC CPU2000 benchmarks with 50% or fewer compulsory misses. Jaleel et al. [37] propose the Re-reference Interval Prediction (RRIP) scheme, which uses set dueling to dynamically choose between static, scan-resistant RRIP and bimodal RRIP. The authors demonstrate an average throughput improvement of 10% over LRU on a range of workloads, and an improvement over LFU.

[36] Back to the Future Akanksha Jain ISCA 2016 [16] Modeling cache performance beyond LRU Beckmann HPCA 2016

[67] Seshadri The Evicted Address Filter PACT 2012 [65] Sartor Comparative Caching with Keep-Me and Evict-Me INTERACT 2005 [70] Wang Using the compiler to improve cache replacment decisions PACT 2002

5.3 Cache Replacement Policy

Motivation: show potential improvement, demonstrate the thrashing occurs

Figure: Cache hit rate, memory bandwidth utilization, and speedup with infinite sized cache / optimal replacement policy

Present different policies and explain tradeoffs (performance vs. speed vs. area vs. power) e.g. evict-me, keep-me, multi-bit priority overlapping vs. non-overlapping replacement groups

Combine with prefetcher or variable granularity memory access??

5.4 Implementation Details

How implemented in simulator

How it could be implemented in HW+SW

5.5 Experimental Results

compare policy to LRU, DIP, RRIP, optimal (Belady)

vary cache size

5.6 Conclusion

Chapter 6

Conclusion

6.1 Conclusion

This paper presents a performance characterization of several parallel graph analysis algorithms on real-world datasets. Our analysis is based on comprehensive experiments from architectural simulation of shared-memory multicore systems. We demonstrate that existing cache and prefetching techniques show limited effectiveness in spite of available data locality. We find that graph algorithm data reuse patterns correlate with graph dataset structure and vertex degree, but cannot be effectively exploited by LRU cache replacement policies. These insights can be leveraged to improve graph analysis algorithm performance.

In this paper, we analyze the performance shortcomings of the conventional FW-BW-Trim algorithm when applied to *small-world* graph instances. We propose three simple extensions to the conventional algorithm that take advantage of *small-world* graph properties to address the deficiencies in existing algorithms. Consequently, our extensions result in significant parallel and sequential performance improvements on *small-world* graph instances to deliver state-of-the-art parallel SCC detection performance.

As a next step, we plan to implement our algorithm in a distributed environment. Our extensions can be easily implemented in such an environment as they only require data from direct neighbors.

6.2 Future Work

fine-grained / variable granularity memory system prefetching communication between threads data vs. computation migration load balance, optimizing parallelization new architecture using upcoming technology

Appendices

Appendix A

Green-Marl Implementation of Graph Kernels

A.1 Breadth-First Search (BFS)

```
7  Procedure BFS(G: Graph, StartNodes: Node_Seq) {
8    Foreach(s: StartNodes.Items) {
9      InBFS(v: G.Nodes From s) {
10     }
11   }
12 }
```

A.2 Single-Source Shortest Paths (SSSP)

A.3 Betweenness Centrality (BC)

```
13 Procedure BC(G: Graph, StartNodes: Node_Seq; BC: Node_Prop<Float>(G)) {
14   G.BC = 0;       // initialize BC
15   Foreach(s: G.Nodes) {
16     // define temporary node properties
17     Node_Prop<Float>(G) sigma;
18     Node_Prop<Float>(G) delta;
19         s.delta = 0;
20     s.sigma = 1; // Initialize sigma for root
21     // Traverse graph in BFS-order from s
22     InBFS(v: G.Nodes From s) {
23       // sum over BFS-parents
24       v.sigma = Sum(w: v.UpNbrs) {w.sigma};
25     }
26     // Traverse graph in reverse BFS-order
```

```
27      InReverse(v!=s) {
28        // sum over BFS-children
29        v.delta = Sum (w:v.DownNbrs) {
30           v.sigma / w.sigma * (1+ w.delta)
31        };
32        v.BC += v.delta @ s; //accumulate BC
33          }
34    }
35  }
```

A.4 PageRank (PR)

```
36  Procedure PageRank(G: Graph, e,d: Double, max_iter: Int, PR: Node_Prop<Double>(G)) {
37    Double diff = 0;
38    Int cnt = 0;
39    Double N = G.NumNodes();
40    G.PR = 1.0; // Init PR
41    Do { // Main Iteration
42      diff = 0.0;
43      Foreach (t: G.Nodes) {
44        Double val = (1-d) + d * Sum(w: t.InNbrs) {
45              w.PR / w.OutDegree() };
46        diff += | val - t.PR |;
47        t.PR <= val @ t; }
48      cnt++;
49    } While ((diff > e) && (cnt < max_iter)); }
```

A.5 Hubs and Authorities (HITS)

```
50  Procedure HITS(G: Graph, max_iter: Int; x,y: Node_Prop<Double>) {
51    Int i = 0;
52    Double x_norm, y_norm;
53    // set inital authority and hub weights to 1
54    G.x = 1.0;
55    G.y = 1.0;
56    // iteratively update weight for each node
57    Do {
58          Foreach (t: G.Nodes) {
59        // I operation: updates x-weights (authority)
60            t.x = Sum(w: t.InNbrs) { w.y };
61      }
62      Foreach (t: G.Nodes) {
63        // O operation: updates y-weights (hub)
64        t.y = Sum(w: t.OutNbrs) { w.x };
65      }
66      // Calculate sum of squares of each weight
67      x_norm = Pow(Sum(a: G.Nodes) { Pow(a.x, 2.0) }, 0.5);
68      y_norm = Pow(Sum(b: G.Nodes) { Pow(b.y, 2.0) }, 0.5);
69      Foreach (t: G.Nodes) {
70        // Normalize
71        t.x = t.x / x_norm;
```

```
72        t.y = t.y / y_norm;
73          }
74      i++;
75    } While (i < max);
76  }
77
78  \section{Conductance (CDT)}
79  %\begin{figure*}[h]
80  \begin{lstlisting}[name=Code]
81  Procedure Conductance(G: Graph, member: Node_Prop<Int>(G), num: Int) : Float {
82      Int Din, Dout, Cross;
83      // count edges inside, outside and cross
84      Din  = Sum(u:G.Nodes)(u.member == num){u.Degree()};
85      Dout = Sum(u:G.Nodes)(u.member != num){u.Degree()};
86      Cross = Sum(u:G.Nodes)(u.member == num){ Count(j:u.Nbrs)(j.member!=num)};
87      Double m = (Din < Dout) ? Din : Dout;
88      If (m == 0)
89            Return (Cross == 0) ? 0.0 : INF;
90      Else
91            Return (Cross / m);
92  }
```

A.6 Connected Components (CC)

Appendix B

Parallel SCC Implementation

Appendix C

Reuse Distance Analysis Implementation

Bibliography

[1] 10th DIMACS implementation challenge. `http://www.cc.gatech.edu/dimacs10/downloads.shtml`. Accessed: 2014-03-07.

[2] Graph 500 benchmark. `http://graph500.org`.

[3] Koblenz network collection. `http://konect.uni-koblenz.de`. Accessed: 2014-03-07.

[4] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. Crono: A benchmark suite for multi-threaded graph algorithms executing on futuristic multicores. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 44–55, 2015.

[5] S. Allesina, A. Bodini, and C. Bondavalli. Ecological subsystems via graph theory: the role of strongly connected components. *Oikos*, 110(1):164–176, 2005.

[6] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A nucleus for a web of open data. In *Proc. Int. Semantic Web Conf.*, pages 722–735, 2008. the DBpedia dataset.

[7] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54, 2006.

[8] D.A. Bader and K. Madduri. Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *IEEE IPDPS*, 2008.

[9] David A Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. Springer, 2014.

[10] A.L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[11] J. Barnat, P. Bauch, L. Brim, and M. Ceška. Computing strongly connected components in parallel on cuda. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 544–555. IEEE, 2011.

[12] J. Barnat, J. Chaloupka, and J. van de Pol. Improved distributed algorithms for scc decomposition. *Electronic Notes in Theoretical Computer Science*, 198(1):63–77, 2008.

[13] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 12. IEEE Computer Society Press, 2012.

[14] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–10, 2012.

[15] Scott Beamer, Krste Asanović, and David Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 56–65, 2015.

[16] Nathan Beckmann and Daniel Sanchez. Modeling cache performance beyond lru. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 225–236. IEEE, 2016.

[17] Bryan T Bennett and Vincent J. Kruskal. Lru stack processing. *IBM Journal of Research and Development*, 19(4):353–357, 1975.

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

[19] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer networks*, 33(1):309–320, 2000.

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

[21] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

[22] Shuai Che, B.M. Beckmann, S.K. Reinhardt, and K. Skadron. Pannotia: Understanding irregular gpgpu graph applications. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 185–195, Sept 2013.

[23] Flavio Chierichetti, Silvio Lattanzi, and Alessandro Panconesi. Rumour spreading and graph conductance. In *SODA*, pages 1657–1663. SIAM, 2010.

[24] F. Chung and L. Lu. Connected components in random graphs with given expected degree sequences. *Annals of combinatorics*, 6(2):125–145, 2002.

[25] Chen Ding and Yutao Zhong. Reuse distance analysis. Technical report, TR 741, Department of Computer Science, University of Rochester, 2001.

[26] L. Fleischer, B. Hendrickson, and A. Pınar. On identifying strongly connected components in parallel. *Parallel and Distributed Processing*, pages 505–511, 2000.

[27] Mark B Gerstein, Anshul Kundaje, Manoj Hariharan, Stephen G Landt, Koon-Kiu Yan, Chao Cheng, Xinmeng Jasmine Mu, Ekta Khurana, Joel Rozowsky, Roger Alexander, et al. Architecture of the human regulatory network derived from encode data. *Nature*, 489(7414):91–100, 2012.

[28] Xiaoming Gu, Ian Christopher, Tongxin Bai, Chengliang Zhang, and Chen Ding. A component model of spatial locality. In *Proceedings of the 2009 International Symposium on Memory Management*, ISSM '09, pages 99–108, 2009.

[29] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. WTF: The who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web*, pages 505–514. International World Wide Web Conferences Steering Committee, 2013.

[30] Saurabh Gupta, Ping Xiang, Yi Yang, and Huiyang Zhou. Locality principle revisited: A probability-based quantitative approach. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 995–1009, 2012.

[31] B. Hendrickson and J.W. Berry. Graph analysis with high-performance computing. *Computing in Science Engineering*, 10(2):14–19, 2008.

[32] R. Hojati, R. Brayton, and R. Kurshan. Bdd-based debugging of designs using language containment and fair ctl. In *Computer Aided Verification*, pages 41–58. Springer, 1993.

[33] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: a DSL for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, 2012.

[34] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. On fast parallel detection of strongly connected components (scc) in small-world graphs. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013.

[35] Sunpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration for multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, 2011.

[36] Akanksha Jain and Calvin Lin. Back to the future: leveraging belady's algorithm for improved cache replacement. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 78–89. IEEE, 2016.

[37] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM, 2010.

[38] Yunlian Jiang, Eddy Z Zhang, Kai Tian, and Xipeng Shen. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Compiler Construction*, pages 264–282. Springer, 2010.

[39] S. Kazemitabar and H. Beigy. Automatic discovery of subgoals in reinforcement learning using strongly connected components. *Advances in Neuro-Information Processing*, pages 829–834, 2009.

[40] Jon M Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

[41] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 611–617, 2006.

[42] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proc. Int. World Wide Web Conf.*, pages 591–600, 2010.

[43] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, volume 12, pages 31–46, 2012.

[44] Ronny Lempel and Shlomo Moran. Salsa: the stochastic approach for link-structure analysis. *ACM Transactions on Information Systems (TOIS)*, 19(2):131–160, 2001.

[45] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187, 2005.

[46] J. Leskovec, K.J. Lang, A. Dasgupta, and M.W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[47] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`. Accessed: 2014-06-01.

[48] A. Lumsdaine, D. Gregor, B. Hendrickson, J. Berry, and J. Guest Editors. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[49] Kamesh Madduri and David A. Bader. Gtgraph: A suite of synthetic random graph generators. `http://www.cse.psu.edu/~madduri/software/GTgraph/`. Accessed: 2014-04-19.

[50] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[51] W. McLendon III, B. Hendrickson, S.J. Plimpton, and L. Rauchwerger. Finding strongly connected components in distributed graphs. *Journal of Parallel and Distributed Computing*, 65(8):901–910, 2005.

[52] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 117–128. ACM, 2012.

[53] Alan Mislove, Hema Swetha Koppula, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Growth of the flickr social network. In *Proceedings of the 1st ACM SIGCOMM Workshop on Social Networks (WOSN'08)*, August 2008.

[54] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 261–270, 2009.

[55] Xing Niu, Xinruo Sun, Haofen Wang, Shu Rong, Guilin Qi, and Yong Yu. Zhishi.me – weaving Chinese linking open data. In *Proc. Int. Semantic Web Conf.*, pages 205–220, 2011.

[56] Petr Novák, Pavel Neumann, and Jiří Macas. Graph-based clustering and characterization of repetitive sequences in next-generation sequencing data. *BMC bioinformatics*, 11(1):378, 2010.

[57] Molly A O'Neil and Martin Burtscher. Microarchitectural performance characterization of irregular gpu kernels. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 130–139, Oct 2014.

[58] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. Technical report, 1999.

[59] Dimitrios Prountzos and Keshav Pingali. Betweenness centrality: algorithms and implementations. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 35–46, 2013.

[60] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 381–391. ACM, 2007.

[61] John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[62] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488, 2013.

[63] Daniel Sanchez. Github: zsim. `http://github.com/s5z/zsim`, jun 2015.

[64] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Computer Architecture (ISCA), 2013 40th Annual International Symposium on*, 2013.

[65] Jennifer B Sartor, Subramaniam Venkiteswaran, Kathryn S McKinley, and Zenhlin Wan. Comparative caching with keep-me and evict-me. In *INTERACT Workshop*, 2005.

[66] N. Satish, C. Kim, J. Chhugani, and P. Dubey. Large-scale energy-efficient graph traversal: a path to efficient data-intensive supercomputing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 14. IEEE Computer Society Press, 2012.

[67] Vivek Seshadri, Onur Mutlu, Michael A Kozuch, and Todd C Mowry. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 355–366. ACM, 2012.

[68] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, 2013.

[69] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[70] Zhenlin Wang, Kathryn S McKinley, Arnold L Rosenberg, and Charles C Weems. Using the compiler to improve cache replacement decisions. In *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pages 199–208. IEEE, 2002.

[71] D.J. Watts and S.H. Strogatz. Collective dynamics of small-world networks. *Nature*, 393(6684), 1998.

[72] Meng-Ju Wu, Minshu Zhao, and Donald Yeung. Studying multicore processor scaling via reuse distance analysis. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 499–510, 2013.

[73] Yuduo Wu, Yangzihao Wang, Yuechao Pan, Carl Yang, and John D Owens. Performance characterization of high-level programming models for gpu graph analytics. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 66–75, 2015.

[74] Qiumin Xu, Hyeran Jeon, and M. Annavaram. Graph processing on gpus: Where are the bottlenecks? In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 140–149, Oct 2014.

[75] Fan Yang and Andrew A Chien. Understanding graph computation behavior to enable robust benchmarking. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 173–178. ACM, 2015.

[76] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, pages 3:1–3:8, 2012.

[77] Liang Yuan, Chen Ding, D Sefankovic, and Yunquan Zhang. Modeling the locality in graph traversals. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 138–147, 2012.