Compressing Natural Graphs and a Practical Work-Efficient Parallel Connectivity Algorithm

Laxman Dhulipala

May 5

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Guy Blelloch

Submitted in partial fulfillment of the requirements for completing an honors thesis.

Keywords: Graph Connectivity, Graph Compression, Deterministic Paralellism

Abstract

Over the past two decades, the explosive growth of the internet has triggered an enormous increase in the size of natural graphs such as social networks and internet link-structures. Processing and representing large natural graphs efficiently in memory is thus crucial for a wide variety of applications. Our contributions are 1) A parallel graph processing framework for representing compressed graphs with significantly fewer bits per edge, and 2) A simple and practical expected linear-work, polylogarithmic depth parallel algorithm for graph connectivity.

Real-world graphs tend to contain a large amount of locality - vertices within a cluster mostly have edges to other vertices in their cluster. We discuss reordering techniques to make vertex labelings reflect locality inherent in the graph. We introduce a framework supporting compressed representations of graph – Cogra – based off of an earlier graph processing framework, Ligra, and show that algorithms operating on compressed graphs in our framework are as fast, or faster than their uncompressed counterparts. The Ligra and Cogra frameworks allow a user to easily implement simple parallel graph algorithms.

In addition to designing simple algorithms, we would like them to be efficient and have good theoretical guarantees. While most algorithms implemented in Ligra and Cogra have good theoretical guarantees, the connected components algorithm does not, and requires $O((V + E)d)$ where d is the diameter. Addressing this need and the lack of implementations of work-efficient connectivity algorithms, we present a simple and practical work-efficient parallel algorithm for graph connectivity that has a work of $O(m)$ and depth $O(\log^3(n))$. The algorithm is based on recently developed techniques for generating low-diameter graph decompositions in parallel. We discuss implementing both the decomposition algorithm and our connectivity algorithm in C++ using CILK+, and show that our connectivity algorithm on 40 cores achieves 18–35 times self-relative speedup, and 10–25 times speedup over the fastest sequential implementation.

Acknowledgments

This work would not have been possible without the incredible amount of help and mentoring provided by a number of people whom I would like to thank now. Omissions appear to be inevitable, and I sincerely apologize to anyone I forget.

First and foremost, I wish to thank my advisor, Guy Blelloch for playing a crucial role in my development as a student. Among many reasons, I wish to thank Guy for teaching me his patient and thorough style when thinking about and analyzing algorithms. I'll always remember the levity that ensued from Guy's remarks on high-degree vertices in graphs.

Of no less importance, I would like to thank Julian Shun, without whom this research would not have been possible. Among many other reasons, I am grateful to Julian for being a wonderful mentor, model and friend.

I wish to thank the many faculty members and graduate students who spoke with me and gave advice on my work. In particular, I would like to thank Gary Miller and Shen Chen Xu for their valuable insight on low-diameter decompositions. Finally, thank you to my family and friends, who endured the late-night musings of a student debugging C.

Contents

Chapter 1

Introduction

Massive graphs are now found in almost every imaginable field. Whether mining data from social-networks, or attempting to calculate relevancy scores over a link-graph from the Internet, graphs are essential objects that must be maintained and manipulated efficiently in memory. In recent decades, the growth of the Internet has spurred a massive increase in the size of graphs that must be represented by applications. With some real-world graphs now having on the order of a hundred billion edges, both industry and academic researchers are heavily invested in finding ways to make algorithms perform well on these new graphs.

Progress in this area comes in many forms. One approach is to painstakingly optimize code in order to produce an experimentally fast algorithm for a problem. Another approach is to consider the problem algorithmically, and present an algorithm with better asymptotic work and span which causes the code to become significantly more performant on large inputs. A major focus of our work is the latter goal: to find deterministic work-efficient parallel algorithms that are both experimentally fast on a variety of inputs and also supported by robust theory.

While we can extract significant performance gains by discovering new, asymptotically faster algorithms for a problem, we can often speed up an entire suite of algorithms by improving the runtime system on which the algorithm runs. Many modern graph algorithms that must be run on massive real-world graphs are run in the distributed-memory setting, but recent work has shown that shared memory machines are often sufficient for processing even the largest real-world graphs. Compared to the distributed-memory setting, shared memory machines often provide lower communication costs. If the graph in consideration can fit entirely into main memory, algorithms running on shared memory architectures are often orders of magnitude faster than their distributed memory counter-parts.

However, the explosive growth in graph sizes has posed a predicament for shared-memory computing frameworks: if the graph in consideration cannot fit in main memory, then the performance of an algorithm on the shared memory machine will suffer from paging and will almost certainly be slower than running the algorithm in a distributed memory framework. As an example of a modern, real-world graph, the largest non-synthetic graph known to us is the Yahoo graph, sporting 6.6 billion directed edges [59], which we symmetrize to obtain a larger graph with 12.9 billion edges. While our machine with 256GB of main-memory can support this machine, this graph occupies close to half of main-memory. With the sizes of web-graphs increasing exponentially from year to year [21], finding a solution which allows these large graphs

to be processed on a single machine, while not replacing the machine every few years is of great interest.

One possible approach which solves the aforementioned problem while not requiring the user to augment a machine with more memory every few years is to *compress* the web-graph. Various compression schemes for graphs have emerged over the past few decades, ranging from somewhat impractical, but near-optimal schemes for compressing particular classes of graphs [24], to compression schemes that provide efficient queries over the compressed data-structures [5]. We are particularly interested in graph-compression that provides efficient access to the adjacency list of a given vertex.

Another way of improving the performance of graph-algorithms that avoids retooling the framework or hardware itself is to improve the algorithm. To this end, we focus on the *Connectivity Labeling* problem, and develop theoretically sound, work-efficient but highly parallel and performant algorithm. Given a graph, $G = (V, E)$ the connectivity labeling problem is to output a labeling L of the vertices in V such that vertices in the same partition have the same label.

We show that our connectivity labeling algorithm requires linear work and polylogarithmic depth. We also experimentally show that it rivals, or often beats existing parallel implementations of the connectivity labeling problem. One of the main issues we overcome is the dearth of theoretical results regarding existing fast implementations of connectivity, which are typically based off of locking and union-find, and are not theoretically work-efficient. We focused on the connectivity labeling problem because of its conceptual simplicity, and relevance in a variety of fields, ranging from VLSI design to image analysis for computer vision.

The contribution of this thesis is the development of a compressed graph processing framework called *Cogra*, and a new linear work, polylogarithmic depth, work efficient connectivity labeling algorithm. Chapter 2 introduces the problem of graph compression, and considers the difficulties encountered when compressing large natural graphs. Chapter 3 discusses the current state of graph processing frameworks, introduces our graph processing framework, Cogra, and presents experimental results comparing Cogra to other frameworks. Chapter 4 introduces the connectivity labeling problem, and discusses a number of historical approaches to connectivity labeling. In Chapter 5 we present our connectivity labeling algorithm, and consider its experimental performance on a suite of real-world graphs. Finally, in Chapter 6 we summarize our results and conclude.

1.1 Preliminaries

In this section, we introduce notation and definitions that will be used throughout the work. We also describe the experimental setup used in both parts of the work, as well as a number of graphs that we test our work upon.

We refer to a graph $G = (V, E)$. If unspecified, G is assumed to be undirected. Graphs are represented in the adjacency array format, where we maintain an array of edges, denoted E, as well as an array of offsets into E, called V. Degrees are implicitly recovered from V, with the degree for vertex i being calculated as $V[i+1]-V[i]$. $V[n-1]$ is set to $|E|$ to ensure correctness.

Our experiments were conducted on a 40-core Intel machine (with hyper-threading), with 4×2.4GHz Intel 10-core E7-8870 Xeon processors (with a 1066MHz bus and 30MB L3 cache), 256GB of main memory, and hugepages of size 2MB enabled. We run all parallel experiments with two-way hyper-threading enabled, for a total of 80 threads. The programs were compiled with Intel's `icpc` compiler (version 12.1.0) with the `-O3` flag using CilkPlus [27] to express parallelism. When running in parallel, we use the command `numactl -i all` to evenly distribute the allocated memory among the processors' caches. The times that we report are based on a median of three trials.

Both the connectivity algorithms and compression framework were tested on a number of graphs. We now describe graphs that are common to both works. Both experiments involved synthetic graphs created using generators from the Problem Based Benchmark Suite (PBBS) [53]. These graphs include the *rMat* graph, *randLocal* graph, and *3D-grid* graph. The rMat graph [14] is a synthetic graph following the power-law degree distribution. The randLocal graph is a random graph where each vertex has edges to five neighbors, chosen at random. The 3D-grid graph is a grid graph where each vertex has 6 neighbors – two along each dimension. We note that the initial orderings created by the graph generators have good locality.

Both works make use of an atomic compare-and-swap operation, supported on most modern machines. The compare-and-swap function CAS(*loc*, *oldV*, *newV*) atomically checks if the value at location *loc* is equal to *oldV* and if so it updates *loc* with *newV* and returns *true*. Otherwise it leaves *loc* unmodified and returns *false*. We use the notation "&" to denote a pointer to a value. For boolean values, we use the value 1 to refer to $true$ and 0 to refer to $false$.

Unless otherwise specified our algorithms are assumed to be running on the CRCW PRAM model.

Chapter 2

Graph Compression

Work on graph compression has been ongoing in various forms since the early 1980's. Turan [56] showed how to succinctly representing planar graphs in $O(n)$ bits, Naor [33] later improved Turan's result to include general unlabeled graphs, and gave a method that was optimal up to constants in $O(n)$. These initial results were not particularly interested in optimizing either the compression algorithms, or the decompression process. Instead, they focused on achieving information theoretic lower bounds on the number of bits needed to represent a class of graphs (for a class C, this is $O(\log(|C|))$ bits to uniquely represent every $c \in C$). In general, for a random graph on n vertices and m edges, we have an information theoretic lower bound of $\theta(m \log(n^2/m))$ [5]. While these approaches can be implemented, due to their focus on achieving information theoretic lower bounds, the results are mainly of theoretical interest.

Another approach is to use *symbol*-based methods, such as Huffman encoding. In symbol techniques, the algorithm replaces symbols in the data being compressed with shorter symbols, which will reduce the number of bits used if the data is not uniformly distributed over the input alphabet. Dictionary techniques such as Lempel-Ziv [60] also replace input symbols with a smaller set of output symbols. Typically, codes are *prefix-free*, i.e. no code-word is a prefix of another code-word. A prefix-free code has a natural interpretation as a binary tree where code-words reside at leaves of the tree, and the path to that leaf represents the code-word in binary. Symbol-based methods, however, fail to exploit the structure of graphs, and are therefore not particularly suited for compressing graphs.

With the explosive growth of the Internet in the early 21st century, both academics and researchers in industry saw the need to implement fast algorithms that iterated over entire web-graphs. In 2002, Randall [45] described the Link Database, which provides fast access to a web-graph of hyperlinks. In order to represent a larger fraction of the original web-graph in memory, they exploited the fact that in a web-graph most links are between nodes sharing the same host-name. For example, examining a text dump of the `http://www.cmu.edu` website, and grepping for `http://`, nearly all of the links extend to other sub-domains residing under `cmu.edu`.

This empirical observation about web-graphs has been the source of a significant amount of research in the past decade. A few years after Randall's description of the Link Database, Boldi and Vigna [8] described the "webgraph framework", which formalized a number of the intuitions introduced by Randall, and other researchers such as Raghavan and Garcia-Molina [44]. We now

adumbrate these two structural properties of web-graphs:

1. **Locality**: Links from a particular node extend mostly to other nodes in the same domain. Concretely, we are more likely to observe links between nodes whose URL's share a long prefix [8].
2. **Similarity**: Consider a lexicographic ordering of the nodes (by URL). Nodes that are close together in this ordering will have highly similar adjacency lists. An intuitive explanation for this empirical observation is that many sites reuse the same navigational structure (be it a sidebar, or header) and blindly copy the same structure onto a vast array of pages, which leads to significant sharing between the adjacency lists of nearby vertices [8].

Boldi and Vigna exploit these properties in two interesting ways. First they use *reference*-coding in order to exploit the similarity in web-graphs. They also use a technique called difference-coding, which was introduced by Blandford and Blelloch [5] to compress adjacency lists. Combining these techniques, their algorithm maintains a sliding-window over the vertices which difference-codes links that are dissimilar from previous links of vertices in the sliding window. If a run of links are identical to a run of links of another vertex, these links are compressed down to an offset.

Ultimately, our techniques and approaches towards graph compression are based off of the work of Blandford and Blelloch [5, 6]. Their approach is to first pre-process the graph into an ordering which embodies locality inherent in the graph. They then apply the aforementioned difference-coding technique in order to compress an adjacency list. We will describe these techniques in more detail in the following section.

2.1 Locality and Compression

The primary factor limiting whether a given graph can be compressed in many modern compression frameworks is the *locality* present in the graph. This is an implicit quality of the graph that can informally be conveyed by the following constraint. If (u, v) is an edge in our graph, then the sets $N(u)$ and $N(v)$ are likely to be highly similar. Related to the inherent locality in a graph is the ordering (labeling of vertices by numbers) in which the graph is presented. An ordering which respects locality present in the graph will have the property that two vertices that share an edge, say (u, v) will have labels that are close together.

Intuitively, we want the ordering to give vertices within the same cluster or component numbers that are close. Having a good ordering on the vertices provides us with a number performance gains, which we now describe:

1. If we are operating in a distributed memory system where we have k machines, we will typically place n/k distinct vertices onto each machine, i.e. the first n/k vertices go on the first machine, the second n/k on the second machine, etc. If the ordering captures the locality inherent in the graph, then we will minimize the amount of inter-node communication happening in a large number of graph algorithms. For example, in traversal type-algorithms that that write values from a frontier, to the frontier's out-neighbors, having an ordering which respects the locality of the graph will result in less communication as most vertices being written to are likely reside on the same machine.

2. Local orderings are also useful on shared-memory systems. Due to main-memory being roughly on the order of the size of the graph for large graphs, algorithms iterating over a vertex's out-neighbors benefit from highly local orderings due to their out-neighbors often lying on the same, or near-by cache-lines. For example, if the cache-line size is 64 bytes, and we are writing into an array of 4-byte integers consisting of vertex data, and vertex u has an out-neighbor to vertex $u+2$, then if the data for u is already cached, we will be able to write to vertex $u+2$ for free.

Even if a graph has a high degree of locality, its original ordering may not adequately capture this locality. We will see that one of the primary ways to generate a better compression ratio using an algorithm is to simply improve the locality present in the graph's ordering. Algorithms that generate better orderings are known as *reordering* algorithms, and typically operate by generating small vertex separators of a graph, and recursively partitioning the graph into smaller and smaller pieces. The algorithm then assigns vertices in the same piece vertex numbers that are close together.

2.1.1 Measures of Locality and Compression

Two useful statistics for measuring the degree of locality of an ordering are the average log cost, and average log gap cost. The ***log cost*** of an edge (u, v) is $\log_2(\|v - u\|)$, that is the logarithm (base 2) of the absolute value of the difference between v and u. The ***average log cost*** of a graph is the average log cost over all edges in the graph, i.e. $(1/|E|) \sum_{(u,v)\in E} \log_2 |u - v|$.

Another statistic which captures how well a graph compresses under a difference encoding scheme is the average log gap cost. Given a graph, $G = (V, E)$, let $v \in V$ have adjacency list $Adj(v) = \{v_0, \ldots, v_{deg(v)-1}\}$ where $v_i \in Adj(v)$ appear in sorted order. The ***log gap cost*** of an edge, (v, v_i), is $\log_2(\|v_i - v\|)$ for $i = 0$, and $\log_2(\|v_i - v_{i-1}\|)$ for $i > 0$. Furthermore, the ***average log gap cost*** is simply the average log gap cost over all $(u, v) \in E$.

2.2 Reordering Algorithms

Suppose we are given a graph G, with some initial ordering on the vertices, which may or may not respect the underlying locality in the graph. A reordering algorithm, A_R takes G and produces a relabeled graph G' such that under some compression scheme C, G' requires less bits per edge than G. We can further tighten our definition of reordering algorithms by borrowing notation introduced by Boldi et al. [10]. They define a *coordinate-free* reordering algorithm as an algorithm that obtains the same number of bits-per-edge with respect to C, given *any* initial ordering of the vertices in G.

We can view compression-free algorithms as simply taking the original ordering of the vertices in G, and applying a random permutation before starting the reordering. Furthermore, a coordinate-free algorithm merely has the property that it only depends on the link-structure of the graph, and not the latent similarity of two vertex numbers given in the original ordering. We now describe a set of reordering algorithms that embody this property:

2.2.1 Traversal Based Orderings

Given a graph G in a random ordering π, our goal is to now find an ordering π^* that embodies some of the locality inherent in the graph. Perhaps the simplest approach to extracting some of the locality is to pick a $v \in V$ at random, and run a breadth-first search. Vertices are assigned labels as they are encountered in the search, with the initial vertex being labeled 0, and the final vertex being labeled $n - 1$. We can also label the vertices using depth-first search.

In practice, both algorithms provide usable results, and do manage to capture some of the input graph's locality. However, theoretically, both algorithms suffer due to their inability to perform well on particular types of graphs. BFS for example, suffers from graphs similar to the one in Figure 2.1. Each layer is assigned contiguous vertex numbers, while a near-optimal solution is found using DFS (each path is labeled contiguously). The DFS-solution is shown using numbers. Similarly, DFS performs poorly on graphs which are similar to the grid-graph in Figure 2.2. While BFS finds a solution that keeps each frontier of the graph starting from the top-left corner labeled contiguously, DFS will only give local-numberings for $1/4$ of the links, as it simply produces a long path in the graph.

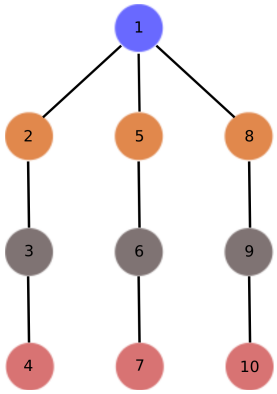


Figure 2.1: An input graph that results in a poor BFS-Ordering. Colors show the ordering produced by BFS, with each frontier having contiguous labels. The numbers show a possible ordering produced by DFS.

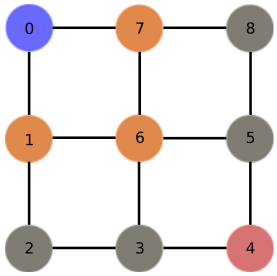


Figure 2.2: An input graph that results in a poor DFS-Ordering. Colors show the ordering produced by BFS, with each frontier having contiguous labels. The numbers show a possible ordering produced by DFS.

We also considered a hybrid-approach, which has the properties of both BFS and DFS. In particular, we visit nodes in DFS order, but label children of each node with consecutive IDs

before recursively calling the function on the children. Empirically, we observed this approach to provide compression-ratios between those of BFS and DFS, as typically, either BFS or DFS produced the best results on a given graph.

Lastly, Blandford et al. [5] proposed a recursive algorithm which uses traversals to compute a separator. The algorithm first performs a breadth-first search from an arbitrary node, and finds the furthest node from the starting node. Then, from this second node, the algorithm performs a second breadth-first search until the search visits half of the vertices. The second search induces an edge-cut of G, between which the vertices are partitioned. It then assigns the range $[0, n/2)$ to one half of the vertices, and $[n/2, n)$ to the second half, and recursively applies the algorithm on the induced subgraphs. In our experiments, we denote this algorithm (bfs-r), meaning recursive breadth-first search.

2.2.2 Graph-Partitioning Algorithms

Historically, the best results for reordering algorithms have come from programs such as METIS [26], which is used for graph-partitioning. Given a graph, $G = (V, E)$, the graph-partitioning problem is to partition V into p almost equal pieces such that we minimize the number of edges between vertices in two different pieces. We will revisit this problem when discussing graph connectivity, as fast, randomized solutions to this problem are essential for our approach to connectivity.

The output of a graph partitioning algorithm for $p = 2$ is a partition of V into two sets, V_1 and V_2 which induces an edge-cut of the graph. As the graph-partitioning problem is NP-Complete for general graphs, algorithms typically rely on heuristic approaches in order to construct good partitions. Historically, approaches have been split into three categories: spectral approaches (based on computing the eigenvector that corresponds to the second smallest eigenvalue), geometric approaches, which associate each vertex with a coordinate, and multilevel partitioning algorithms, which recursively contract and uncontract the graph to recover a cut. The algorithms we compare against fall into the latter category, and we will briefly touch on how they operate.

METIS [26] for example uses the following simple algorithm to compute a 2-partition of the graph. They solve the general case of a k-partition by first computing a 2-partition, and then recursively subdividing each of the two vertex sets into more pieces if necessary. The graph is first coarsened into a sequence of graphs, $G_1, G_2, \ldots, G_j$ where $|V_i| > |V_l|$ for $i < l$. This process stops once the number of vertices is less than some constant threshold, at which point, they compute a near-optimal bisection of this small graph. They then undergo j refinement steps, one for each coarsened graph, where the bisection for G_{i+1} is projected onto G_i, and adjusted using a heuristic. The subtlety in METIS, and other recursive schemes is in the choice of coarsening algorithm, and the choice of the refinement algorithm used in the uncoarsening phase.

We also compare our schemes against Scotch [38] and PT-Scotch [15], a parallel version of Scotch. Both frameworks operate on the same recursive bisection scheme as METIS. We found that the compression quality of PT-Scotch to be very similar to METIS, and therefore only provide experimental evaluation against METIS.

Lastly, the Problem Based Benchmark Suite (PBBS) [53] includes a recently developed parallel separator-based reordering algorithm, which we call p-sep. It is very similar in design to a sequential algorithm designed by Blandford et al. [5]. The Blandford algorithm contracts along

edges until a single vertex remains. They introduce the following metric for choosing which edge to contract: $w(E_{AB})/(s(A)s(B))$ where $w(E_{AB})$ is the weight of the edge between vertices A and B and $s(A)$ and $s(B)$ are the weights of A and B, respectively.

The initial weights of all vertices and edges are all 1. As two vertices are contracted together, the super-vertex is assigned a weight of $s(A)+s(B)$. When a multi-edge results from contracting two vertices, the new edge is assigned the sum of the weights of the previous edges. The vertex-labeling is an in-order traversal of the leaves of the separator tree. The implementation in PBBS produces a parallel algorithm which follows roughly the same idea. They extract parallelism by ensuring that each vertex only participates in at most one contraction. For each vertex, they choose an edge which maximizes the previously described metric, which induces a forest over the original graph. They then apply a parallel maximal matching algorithm on the forest in order to determine which edges are contracted.

2.3 Difference Coding and k-bit Codes

The ***difference encoding*** compression scheme takes a vertex, v and $Adj(v) = \{v_0, v_1, \ldots, v_{deg(v)-1}\}$ where $Adj(v)$ is in increasing order, and encodes the differences, $\{v_0 - v_1, v_1 - v_0, \ldots, v_{deg(v)-1} - v_{deg(v)-2}\}$. Notice that the average log gap cost defined in section 2.1.1 captures the average $\log_2$ of the differences being encoded. We can encode these differences using a variety of codes. Choosing a code forces us to make a trade off between the number of bits per link, and the ease of decoding encoded values. We restrict our choice of codes to prefix-free logarithmic codes, which we now describe.

A ***logarithmic code*** is a prefix code which uses $O(\log x)$ bits in order to represent a number, x. Perhaps the best known logarithmic code is the *gamma code* [18], which represents an integer, x as $\lceil \log_2 x \rceil$ in unary, followed by $x - 2^{\lceil \log_2 x \rceil}$ in binary, using a total of $1 + 2\lfloor \log_2 x \rfloor$ bits. Gamma codes are particularly useful when the size of the number being encoded is unknown. However, unless the numbers being decoded are in a small range, decoding gamma codes is often prohibitively expensive. For encoded numbers in a small enough range, the decoding can be accelerated by using table look-up.

Blandford et. al [6] describe another class of codes, known as k-bit codes, which encode an integer x as a series of k-bit blocks. Each block uses one bit as a *continue bit*, which indicates if the following block is also a part of x's encoded representation. To encode x, we first check if $x \leq 2^{k-1}$. If this is the case, we simply write the binary representation of x into a single block, and set the continue bit to 0. Otherwise, we write the binary code for $x \mod 2^{k-1}$ in the block, set the continue bit to 1, and then encode $\lfloor x/2^{k-1} \rfloor$ in the subsequent blocks. Decoding works by examining blocks until a block with a continue bit of 0 is found. If b blocks are examined, then the decoded value in the i'th examined block is multiplied by $2^{(b-i)(k-1)}$, and added to the result.

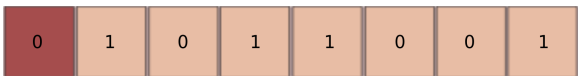


Figure 2.3: The value 89 encoded using byte-codes using 1 byte, or 8 bits

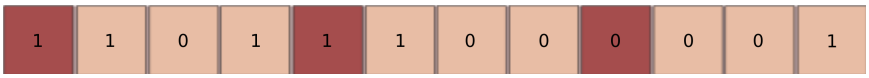


Figure 2.4: The value 89 encoded using nibble-codes using 3 nibbles, or 12 bits

We use two types of k-bit codes in our experiments - byte-codes and nibble-codes, which correspond to an 8-bit and 4-bit code respectively. Byte-codes are extremely fast to decode, as compressed blocks lie on byte-aligned boundaries. Nibble-codes lie on 4-bit boundaries, require more memory accesses and are slower to decode. 4-bit and even 2-bit codes often provide better compression if most values being encoded lie off of bit boundaries that causes the code to use an extra block to store very few bits (with respect to k).

Lastly, notice that in our difference encoding scheme, the first value we encode in an adjacency list may be negative, as $v_1 - v$ may be negative. Therefore, we store an extra sign bit for the first value.

Chapter 3

A Compressed Graph Processing Framework

There are a large number of graph processing framework available today, ranging from frameworks designed for distributed memory systems, to frameworks capable of processing massive real-world graphs on a single commodity machine. Different frameworks also come packaged with vastly different features, ranging from built-in machine learning algorithms, to functionality for dynamically modifying the graph. Furthermore, frameworks differ vastly in the syntax and semantics they impose on the programmer. We first describe the current state of graph processing frameworks, identify areas of focus, and finally describe and evaluate Cogra, a compressed graph processing framework for shared-memory.

Recent frameworks can be broken down into roughly the following categories:

1. **Vertex-Centric Models**: Frameworks using the vertex-centric model have the programmer write functions from the perspective of a vertex. Each vertex is allowed to iterate over its edges, and write messages to its neighbors. Historically, these frameworks were bulk-synchronous [20, 30, 48], but other frameworks such as GraphLab have introduced asynchronous computation, which allows for fast machine learning algorithms on graphs.
2. **Graph-Centric Models**: Recently, a framework called Giraph++ [55] proposed a programming model which exposes information about partitioning information to the application programmer in order to take advantage of algorithm-specific optimizations. They advocate this system due to the inability to implement these optimizations in vertex-centric models such as GPS, Pregel, and the original Giraph framework. By exposing subgraph information, a vertex can effectively 'look past' its neighbors, and pass information to all vertices within its partition.
3. **Matrix-vector and Matrix-Matrix Models**: These frameworks provide primitives for sparse matrix-matrix, and sparse matrix-vector computations. They include efficient implementations of linear algebra primitives that then operate on graphs represented as matrices. A notable example is the Combinatorial BLAS [13], which is sometimes used as a backend for other frameworks such as the Knowledge Discovery Toolbox [29].
4. **Domain-Specific Models**: Ligra [51] is a recent lightweight framework supporting simple expression of algorithms based on graph traversals, such as Breadth First Search and

PageRank. It can run these algorithms on the Yahoo graph, which is the largest publicly available real-world graph in speeds that are orders of magnitude faster than speeds attained from distributed-memory graph processing frameworks. Ligra can be broadly classified under the vertex-centric model category, but due to its specific focus on allowing for the easy implementation *traversal* problems, we classify it as a domain-specific model.

3.1 A Survey of Popular Frameworks

Due to different frameworks often offering completely different features, we describe each framework individually, and finally provide a general critique on the benefits and issues faced when working in these popular frameworks.

The vast majority of graph frameworks in the past decade have been centered around either distributed memory graph-processing, or MapReduce based models[25, 28, 29, 30, 48]. Pegasus [25] is a library designed for computing Petabyte scale graphs using the Hadoop implementation of MapReduce. As it is built on top of MapReduce, it is difficult to make it highly performant due to the large amount of communication and inter-machine IO. Pegasus' processing model is based off of sparse and generalized matrix operations, but does not allow a sparse representation of the vertices, and is thus inefficient when very few vertices are active.

The Knowledge Discovery Toolbox (KDT) [29] operates using sparse and generalized matrix operations, and bases its core library off of the Combinatorial BLAS. Their framework allows for both sparse vectors and sparse matrices, and can efficiently support only a small subset of vertices being active during an algorithm. However, they do not currently support switching between dense and sparse representations of a vertex set based on the set's density.

Pregel [30] is a vertex-centric graph processing framework in the distributed setting. Functions are written from the perspective of a vertex, which is able to iterate over its edges and send messages to neighbors. The message passing is bulk-synchronous, making computed values only appear in subsequent rounds. Due to operating on distributed memory machines, Pregel is not highly performant. GPS [48] implements the Pregel interface and also supports graph partitioning and computation reallocation, but despite this only achieves a marginal speedup relative to Pregel.

GraphLab [28] is an asynchronous framework for parallel graph processing, and supports both shared-memory and distributed-memory machines. GraphLab's vertex centric functions can run at any time, as long as a set of specified consistency rules are obeyed. This makes the framework particularly powerful for machine learning algorithms operating on graphs, such as topic modeling. Both Pregel and GraphLab assume a single graph, with values stored at nodes - a framework imposed limitation which ultimately restricts the types of applications expressable by the framework.

Grappa [34] is a runtime system which supports scalable support for irregular parallel algorithms. Grappa works for a number of applications, including branch-and-bound optimization, circuit simulation, and graph processing. Irregular applications are difficult to scale due to large and unpredictable amounts of communication, and very poor data locality. In order to resolve this difficulty, Grappa allows high-latency in communication for a large increase in total network throughput. Instead of sending messages as they appear, they aggregate and batch messages in

various parts of the system. They show that their framework running on commodity hardware is competitive against hand-optimized MPI code, and to code running on the Cray XMT machine.

There are also a number of shared-memory frameworks, including Grace [43], X-Stream [47], GRACE [57, 58], and Galois [36, 41].

Ultimately, while frameworks that are designed to run on distributed memory machines can in theory support massive peta-byte sized graphs [25, 30], applications running on-top of these frameworks are severely limited due to the sheer limitations of inter-node communication and inter-node IO. Furthermore, a number of frameworks impose semantic restrictions, forcing the programmer to specify a *single* function for iterating over vertex's out-edges. Finally, frameworks such as GraphLab and Pregel provide the user with no way of representing multiple sets of vertices, and iterating over them simultaneously. We now turn our discussion to the Ligra graph processing framework, which addresses a number of these issues.

3.2 The Ligra Graph Processing Framework

Our compressed framework is built on top of *Ligra*, a graph processing framework developed by Julian Shun and Guy Blelloch. Ligra is specific to shared-memory machines, and provides primitives designed to make graph traversal algorithms very easy to express. In this section, we summarize the key features and primitives of Ligra, and also discuss a key optimization implemented in Ligra that significantly improves the performance of graph traversals.

Ligra provides the user with two data-types - a ***graph*** datatype, representing $G = (V, E)$, and a ***vertexSubset*** datatype, representing a subset, $V' \subset V$. Ligra also provides the application programmer with functions for iterating over a graph in frontiers. The first function is a ***vertexMap***, which allows the user to map over vertices, and the second function is a ***edgeMap***, which allows the user to map over edges. The library also provides standard functions for querying the size of datatypes, as well as constructors.

Internally, Ligra represents graphs using the compressed sparse row (CSR) format. A directed graph is stored as two arrays, one storing the in-edges of vertices, and the other storing out-edges. An array of vertices is also maintained, with each vertex storing pointers into the in-edge and out-edge arrays, along with the in-degree and out-degree. As the CSR format for inputting graphs to Ligra can only represent directed graphs, when passing an undirected graph into Ligra, the symmetric flag must be passed. For symmetric graphs, only one array of edges is stored.

The ***vertexMap*** function takes as input a vertexSubset, U, as well as a boolean function F, applies F to all $u \in U$, and returns the set $U' = \{u \in U | F(u) = 1\}$. It has the type

$$\text{vertexMap} : (U : \text{vertexSubset}, F : \text{vertex} \mapsto \text{bool}) \mapsto \text{vertexSubset}$$

The ***edgeMap*** function takes as input a graph, $G = (V, E)$, a vertexSubset, $U \subset V$, a boolean function on edges, F, and a boolean function C on vertices. The function takes all $(u, v) \in E$ where the source vertex, $u \in U$, and checks $F(u, v)$. If this is true, it then checks $C(v)$. It then returns all vertices v satisfying both properties. Succinctly, it returns a set of vertices $V = \{(u, v) \in E | u \in U, F(u, v) = 1, C(v) = 1\}$. Intuitively, the edgeMap function allows the user to specify a graph traversal by specifying two functions - F, and C, where F is a function specifying a condition on edges necessary for the target to be included in the next

frontier, and C is a function specifying a condition on vertices necessary for the target to be included in the next frontier. It has the type

$$\text{edgeMap} : (U : \text{vertexSubset}, F : \text{vertex} \times \text{vertex} \mapsto \text{bool}, C : \text{vertex} \mapsto \text{bool}) \mapsto \text{vertexSubset}$$

Ligra also makes use of the ***compare-and-swap*** (CAS) operation, which is described in Section 1.1. Compare-and-swaps are used in a number of applications implemented in the framework. Ligra also provides a diverse set of applications implemented in the framework, including breadth-first search (BFS), Betweenness Centrality (BC), Radii Estimation (Radii), Connected Components (CC), and lastly PageRank.

3.2.1 Dense and Sparse Representations of Frontiers

While Ligra exposes a single edgeMap function to the application programmer, it has two private implementations of edgeMap not visible to the programmer. The first version is a sparse version of edgeMap, which is used when the size of the vertexSet being iterated over is small. The second version is a dense representation, which is used when the size of the vertex set is large.

The sparse edgeMap is effectively a *write*-based method for building a new frontier - each vertex in the current frontier writes to its out-neighbors, applying the F and C functions described above to decide whether to include its neighbor in the next frontier. Because the size of the frontier is smaller than a threshold, the total work is bounded by the sum of the out-degrees of frontier vertices.

On the other hand, the dense edgeMap can be viewed as a *read*-based method for building a new frontier. Every unvisited vertex in the graph satisfying $C(v)$ will iterate over its out-neighbors, checking to see if any of them are in the current frontier. The dense method can be performed in parallel over all vertices, skipping over a vertex if it is already visited. Furthermore, once a vertex i being investigated in the dense version has been added to the vertex-set, it can stop iterating over its out-neighbors, which makes this method more efficient than edgeMapSparse if many vertices in the current frontier have edges to i.

This optimization implemented in Ligra is based off of the work of Beamer et al. [3, 4], who worked on developing a highly performant breath-first search algorithm for shared-memory machines. Their result was a hybrid technique which they called a "Direction-optimizing breath-first search". They introduce the idea of a "bottom-up" BFS (in contrast to the top-down BFS, which simply takes the current frontier, and in the worst-case inspects every out-edge), which is used when the size of the current frontier is large. The bottom-up method is significantly faster when the size of the current frontier is large because for a given unvisited vertex in the graph, it will stop iterating over its out-edges once it has found an edge to a vertex in the current frontier.

3.2.2 The Need for Compression

While Ligra addresses a number of issues raised regarding the current state of graph-processing frameworks, it fails to address the issue of keeping up with ever-increasing scale of modern graphs[21]. Compared to distributed memory frameworks, which can simply throw more machines at an enormous graph, shared-memory systems simply cannot fit the entire graph in mem-

ory, and as a result have to resort to paging, or processing the graph entirely from disk, which typically cripples the performance of graph algorithms.

Furthermore, even high-end modern shared-memory machines, such as the Intel Sandy Bridge based Dell R910 which has 32 cores (64 hyper-threads) and can be configured with up to 2 Terabytes of memory have a fairly large, but finite amount of main-memory. In the not-too-distant future, graphs exceeding this size are likely to become the norm in both industry and academic settings, and handling such graphs elegantly in a shared-memory framework is important for creating performant real-time applications. Compression is crucial for reducing the space of these graphs down to a manageable size, and for fitting the entire graph in memory.

We note that all current publicly available real-world graphs fit in main memory on a single shared-memory machine. Despite this fact, compression is still a useful feature for shared-memory graph-processing frameworks due to the reduced memory footprint of the graph in memory. Furthermore, by compressing the graph, one can use a smaller, and cheaper machine in order to process the graph. Lastly, despite the fact that applications must uncompress the graph in order to access a vertex's adjacency list, we will show that there is little to no performance degradation, and in some cases, performance improvement when running an application on a compressed graph.

3.3 Supporting Compressed Representations

Cogra, our compressed graph processing framework extends Ligra, adding support for compressing existing graphs, and operating on already compressed graphs. Our goal when modifying Ligra was to make no change in the interface presented to the application programmer. To this end, the compression framework resides solely in the backend of ligra, and requires the modification of the private edgeMap functions.

3.3.1 Encoding

Cogra provides an *encoding* program that takes a graph given in compressed sparse row format and generates a binary file of the compressed graph. Adjacency-lists are compressed using the difference-encoding technique described in Chapter 2. Each difference-encoded adjacency list is then encoded using a k-bit codes. We implemented two k-bit codes - byte, and nibble codes, which are 8-bit and 4-bit codes, respectively. The encoder emits a graph in binary, which consists of an array of vertices, followed by two arrays of compressed in-edges and out-edges. If the graph is specified to be symmetric, as is the case for undirected graphs, a single array of edges is written.

Byte-codes are fast to decode, as they lie on byte-aligned boundaries. However, if most values being encoded lie between $[0, 2^3)$, then nibble-codes are likely to provide significantly better compression. Nibble-codes lie on 4-bit boundaries, and require more bit-operations to decode. Our implementation of nibble-encoding places the first nibble on a byte-aligned boundary. This makes accessing the start of a vertex's compressed adjacency list significantly easier, as we simply store a pointer to the start of the list, instead of an offset from a base consisting of the number

```
1: procedure PARALLELCOMPRESS(G = (V,E))
2:     C = alloc(|V|)
3:     Adjs = alloc(|V|)
4:     parfor v ∈ V do
5:         (Adjs[i], size) = sequentialCompress(v,v.deg,&Adjs[i])
6:         C[i] = size
7:     totalSize = plusScan(C, C, |V|)
8:     return compact(Adjs, C, alloc(totalSize))
```

Figure 3.1: parallelCompress implementation

```
1: procedure SEQUENTIALCOMPRESS(v, deg, &outEdges, &Out)
2:     offset = 0
3:     if deg > 0 then
4:         offset = CompressFirstEdge(&outEdges[0], v, &Out, offset)
5:         for j = 0 to deg−1 do                          ▷ Loop over out-neighbors
6:             offset = CompressNextEdge(&outEdges[j], &outEdges[j-1], &Out, offset)
7:     return offset
```

Figure 3.2: sequentialCompress implementation

of nibbles before the start of the adjacency list. In practice, we found byte-aligning the first nibble to require minimal extra space.

We compress the entire edge-set in parallel, using a function *parallelCompress*, which we now illustrate here.

ParallelCompress exploits parallelism over the vertices, sequentially compressing each vertex's adjacency list in parallel. Each vertex writes its compressed adjacency list representation into memory separately, storing a pointer in Adjs, and returning a size denoting the total number of bytes used. It then performs a parallel prefix-sum (plusScan) using the associative operator $+$ in order to determine the total size of the compressed edge-array. The final step is performed by *compact*, which in parallel, copies the adjacency lists stored in Adj into one contiguous block of memory.

Notice that we cannot easily compress a given vertex's adjacency list in parallel. This is due to the fact that the size of the compressed representation of a given difference, such as $v_i - v_{i-1}$ is unknown - it could take a single byte, or two bytes, or possibly in the worst case, more bytes than an un-compressed representation of the difference. Because of this, we compress each adjacency list sequentially while maintaining an offset into the compressed-adjacency list.

In the pseudo-code for sequentialCompress, the bulk of the work is done by two functions, CompressFirstEdge, and CompressNextEdge. CompressFirstEdge takes two values - v_0, the first vertex in v's sorted adjacency list, as well as v. It then stores a k-bit encoded version of the difference, $v_0 - v$. Because this quantity may be negative, we store the absolute value, $|v_0 - v|$ and also use an extra-bit which stores the sign of the difference. The second function, CompressNextEdge simply stores the difference $v_i - v_{i-1}$. As the adjacency list is sorted, and we do not support multi-edges, all differences will be strictly greater than 0.

Once a graph has been processed by the encoder program, its compressed representation is written to disk. Because the compression must be performed a single time (the Cogra runtime

```
1:  procedure DECODESPARSE(v, deg, &outEdges, F, C, &Out)
2:      prevEdge = −1
3:      for j = 0 to deg−1 do                          ▷ Loop over out-neighbors
4:          if j = 0 then
5:              ngh = FirstEdge(&outEdges) + v
6:              prevEdge = ngh
7:          else
8:              ngh = NextEdge(&outEdges) + prevEdge
9:              prevEdge = ngh
10:         if (C(ngh) == 1 and F(v, ngh) == 1) then
11:             Add ngh to &Out
```

Figure 3.3: decodeSparse implementation

simply reads the compressed representation into memory), we did not pursue minute optimizations regarding the efficiency of our compression.

3.3.2 Decoding

In order to support decoding, we modified several pieces of Ligra, none of which alter the interface observed by the application programmer. Firstly, the function used to construct a graph from a file was altered to take as input a compressed graph (possibly symmetric). It simply reads in the binary compressed graph, writes it into memory, and returns a pointer to the graph.

The interfaces to edgeMap, and vertexMap provided by the framework are identical to those of Ligra. However, the private functions called by edgeMap, namely edgeMapSparse and edgeMapDense are modified to iterate over the compressed graph. As the graph is accessed primarily for traversal-type queries, we implemented a generic function, *decode*, which decodes a given vertex's adjacency list. For the purpose of illustration, we will describe two functions: decodeSparse and decodeDense, which correspond to versions of edgeMapSparse and edgeMapDense that decode the compressed graph. The corresponding edgeMapSparse and edgeMapDense functions in Cogra are effectively wrappers around the two decoding functions.

DecodeSparse takes as input a vertex v, its degree, denoted as deg, a pointer $\&outEdges$, which is the location in memory of the start of vertex v's compressed adjacency list, as well as the functions F and C. Recall that F is a boolean function over edges, and C is a boolean function over vertices. The pseudo-code for decodeSparse is more complicated than that of edgeMapSparse due to the logic for decoding the compressed adjacency list. We use two functions to decode compressed values, *FirstEdge* and *NextEdge*.

The FirstEdge function takes as input a memory location, $\&loc$, and decodes the k-bit encoded value stored at $\&loc$. It also decodes the specially stored sign-bit for the first value, as the first value stored may be negative. For all other values stored following the first compressed value, the sign-bit is not stored, as values are strictly greater than 0. The NextEdge function simply decodes these k-bit encoded values. It requires the value of the previous edge, as the value stored is the difference $v_i - v_{i-1}$. We implemented two versions of FirstEdge and NextEdge, one for byte-codes, and one for nibble-codes respectively.

DecodeSparse then iteratively decodes the entire adjacency list. For each decoded edge, we

```
1:  procedure DECODEDENSE(i, deg, &inEdges, F, C, &Out, U)
2:      prevEdge = −1
3:      for j = 0 to deg−1 do                          ▷ Loop over in-neighbors
4:          if j = 0 then
5:              ngh = FirstEdge(&inEdges) + v
6:              prevEdge = ngh
7:          else
8:              ngh = NextEdge(&inEdges) + prevEdge
9:              prevEdge = ngh
10:         if (ngh ∈ U and F(v, ngh) == 1) then
11:             Add i to &Out
12:         if (C(i) == 0) then break
```

Figure 3.4: decodeDense implementation

check to see whether $F(v, ngh) == 1$, and if the target, ngh, satisfies $C(ngh) == 1$. If this the case, then we add ngh to Out, the vertexSubset that is returned. Notice that we must maintain the value prevEdge, storing the target vertex of the previous edge, as the PrevEdge function requires the value of the previous edge in order to decode the current edge.

DecodeDense takes as input a vertex, i, which is not in U, i's degree - deg, a pointer to i's in-edges, &inEdges, as well as F, C and &Out, a pointer to the new frontier returned by edgeMapDense. It also takes U, the current frontier set. The functions F and C are specified identically to the functions used in decodeSparse. DecodeDense then iterates over i's in-edges, decoding the first edge using FirstEdge, and subsequent edges using NextEdge. Once again, the id of the previously decoded vertex is stored in prevEdge for use in NextEdge.

If the decoded vertex, ngh, which has an in-edge to i is in U, that is $ngh \in U$, and $F(ngh, i)$ then we add i to &Out, placing i in the new frontier. Implicit in decodeDense is the fact that $C(i) == 1$ - otherwise, we would not have bothered iterating over its in-edges to check if it should be a part of the new frontier. Notice that in line 12, we perform if$(C(i) == 1)$then break. This is done in order to break out of the loop early, and stop iterating over the subsequent out-neighbors if i is already added to Out.

Notice that deg, the degree of v is passed into both functions. We also tested an implementation where the degree was also k-bit encoded into a vertex's adjacency list, but found this version to provide no substantial speedup.

3.4 Applications

We test our implementation on the same applications as Ligra in order to provide a comparison of the cost of compression in a graph-processing framework. We now describe the five applications as they are implemented in Ligra. Because Cogra maintains the same interface for the application programmer as Ligra, the applications were not modified at all.

Breadth-First Search.

Ligra implements the top-down version of BFS. Given a graph, $G = (V, E)$, and a starting vertex, r, (this is the first vertex in the ordering, to maintain consistency with Ligra), the algo-

rithm computes the breath-first search tree rooted at r. The algorithm takes as many rounds as the distance farthest away from r, where distance is measured as shortest-path distance on the graph.

Initially, the algorithm creates a single vertexSet, which contains only the root, r. While this vertexSet (representing the current frontier) is non-empty, it performs an edgeMap, which uses an update function that atomically visits unvisited neighbors using a compare-and-swap, and adds them to the next frontier (this is the F function on edges). The C (or Condition) function just checks whether a given vertex has been visited. It is also used to provide an early-termination condition if the framework is using edgeMapDense.

Betweenness Centrality. Ligra also implements an algorithm for computing the betweenness centrality of a vertex. The betweenness centrality of a vertex, v, measures how central, or important the node is in a graph by measuring the number of pair-wise shortest paths between vertices in the graph that pass through v. Formally, given a graph, $G = (V, E)$, let σ_{st} denote the number of $s - t$ shortest paths in G. Let $\sigma_{st}(v)$ denote the number of shortest paths in G passing through v. Then, the betweenness centrality of v is simply the ratio of these two quantities, summed over all $s, t \neq v$, that is:

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

The betweenness centrality algorithm is an implementation of Brandes' algorithm [11], which solves the betweenness centrality by solving the sub-problem of computing single-source shortest-path information from every vertex. Implementing the algorithm for all vertices requires computing two breath-first searches for every $v \in V$, the first operating on the original graph, and the second operating on the transpose of G. As transpose is implemented by simply swapping pointers between the in-edges and out-edges under-the-hood, this operation is not computationally expensive. However, Ligra only performs this computation for a single vertex, and allows the user to run it on more vertices if desired. The Ligra implementation computes an approximate betweenness centrality, and is described in more detail in [51].

Radii Estimation. Ligra provides an implementation of approximate radii calculation. The radii of a given vertex, v, in the graph is intuitively the locally observed diameter of the graph from v. Concretely, given $G = (V, E)$ the Graph Radii problem is compute for each $v \in V$, $\max_{u \in V, u \neq v} d(v, u)$ where $d(u, v)$ is the shortest-path distance between u and v in G. Intuitively, this is the maximum shortest path distance between v and any other $u \in V$. Finally, notice that the diameter of G is simply $\max_{v \in V} Radii(v)$.

A simple and direct implementation to calculate the Radii of G requires solving the single-source shortest path problem for each $v \in V$, which can be done by running a breadth-first search, or another analogous shortest-path algorithm. Because this is fairly computationally expensive, Ligra implements a method for computing an approximation of Graph-Radii using bit-parallelism. Initially, the chooses K vertices from V at random, and assigns all $v \in V$ a bit-vector of length K. The K chosen vertices (labeled $[0, k - 1]$ all flag exactly one bit in their vector to 1). The algorithm then forms a vertexSubset out of these k vertices, and runs a parallel BFS (multiple-source BFS). When processing an edge (u, v) from the current frontier, u bitwise-ORs its bit-vetor with v's vector. If v's bit-vector changes, the v is added to the next frontier. The algorithm terminates (the frontier becomes size 0) when no bit-vectors change between rounds.

Connected Components. We now describe a traversal-based implementation of connectivity labeling. Although we will cover connectivity in great-detail in later portions of this work, we present a simple definition of the problem here. Given an undirected graph, $G = (V, E)$ the connectivity labeling problem is to produce a set of labels, $L, |L| = |V|$, s.t. all vertices in the same component (vertices reachable from each other) have the same label, i.e. $L[u] == L[v]$ if there exists a u, v path, and $L[u] \neq L[v]$ if no u, v path exists.

Ligra implements a simple label-propagation algorithm which works as follows. Initially, every vertex is assigned its own vertex-id as its label, and all vertices are placed on the current frontier. While the current frontier is non-empty, we continue to run an edgeMap which checks for a given edge, (u, v) whether u's number is smaller than v's. If this is the case, it atomically updates v's number to be u, and places v on the next frontier. The algorithm reaches a fixed-point and terminates when every vertex is labeled with the minimum label that is reachable in the graph. Atomically writing the minimum ID is done by using a compare-and-swap [54], called writeMin.

PageRank.

Ligra also provides an implementation of the PageRank algorithm. PageRank is an iterative method to compute the relative importance of nodes in a graph, originally developed to run on a web-graph, and compute the importance of webpages [12]. Concretely, given a graph $G = (V, E)$, and a damping factor $0 \leq \gamma \leq 1$, and a constant ϵ, the algorithm initializes each vertex's rank to be $1/|V|$. On each round, it applies an update rule which modifies a vertex's rank as follows:

$$\mathrm{R}[v] = \frac{1-\gamma}{|V|} + \gamma \sum_{u \in N^-(v)} \frac{\mathrm{R}[u]}{deg^+(u)}$$

where $R[v]$ denotes the rank of v. The algorithm stops updating values for vertices once

$$\sum_{v \in V} R_t[v] - R_{t-1}[v] < \epsilon$$

The Ligra implementation of PageRank simply implements this basic algorithm. Because each vertex is updated in every round, the frontier is always of size $|V|$. This means that the edgeMapDense will always be used (as opposed to edgeMapSparse), which makes the update-step work-optimal, as all vertices will always satisfy $C(i)$ (we only set $C(i) = 1$ once we have reached the termination condition based on ϵ).

3.5 Experiments and Evaluation

We now evaluate both the performance, as well as the compression ratios achieved by Cogra. We test two versions of Cogra, one using byte-codes, and the other using nibble-codes (8-bit and 4-bit codes, respectively). We also perform an evaluation of the compression ratios achieved by our compression algorithms, and evaluate a diverse set of reordering algorithms in order to investigate which reordering algorithm is most effective in reducing the number of bits per edge. We refer the reader to Section 1.1 for an explanation of our experimental setup.

We tested our framework on a suite of both real-world and synthetic graphs. We remind the reader that the synthetic graphs used in our tests are created by graph generators from PBBS and exhibit good locality. We also obtained a suite of real-world graphs. These include the *Yahoo* graph [59], which is the largest non-synthetic publicly available web-graph and is provided by Yahoo. Some graphs, such as *nlpkkt*, are large matrices taken from optimization problems (computing KKT-conditions), and turned into graphs [49]. Our graphs are taken from the Stanford Network Analysis Project (SNAP), and the Florida Sparse Matrix Collection [1, 17]. The *Twitter* graph is a (now slightly old) publicly available graph of the Twitter social network. Finally, the *uk-union* graph is a graph constructed from the union of 12 snapshots of subsets of the United Kingdom web network [9]. We note that both the Twitter and uk-union graphs are asymmetric. We also symmetrized the Yahoo graph in order to construct an even larger graph to test on. Some graphs also include self-loops and multi-edges – these were pruned before compression. All graphs and their respective sizes are shown in Table 3.1.

Input Graph	Num. Vertices	Num. Directed Edges
random	10,000,000	98,201,048
rMat	16,777,216	99,445,780
3D-grid	9,938,375	59,630,250
soc-LiveJournal	4,847,571	85,702,474
cit-Patents	6,009,555	33,037,894
com-LiveJournal	4,036,538	69,362,378
com-Orkut	3,072,627	234,370,166
nlpkkt240	27,993,601	746,478,752
Twitter	41,652,231	1,468,365,182
uk-union	133,633,041	5,507,679,822
Yahoo	1,413,511,391	12,869,122,070

Table 3.1: Graph inputs.

3.5.1 Reordering Algorithms

We now evaluate a collection of reordering algorithms described in Section 2.2. We measure the algorithms performance with respect to the average log cost, and average log gap costs described in Section 2.1.1, as these measures are directly related to the number of bits-per-edge obtained by difference encoding. The reordering algorithms in Table 3.2 include the BFS, DFS, hybrid, recursive, and parallel-separator algorithms described in Sections 2.2.1 and 2.2.2 as well as METIS, which was described in Section 2.2.2.

We do not apply the reordering algorithms on the synthetically generated graphs, due to the graph generators emitting the graph in a local ordering. Furthermore, for some graphs, the parallel-separator (p-sep) algorithm and METIS take too much memory or an inordinate amount of time, and as a result we were unable to obtain any compression results for these ordering algorithms on both the uk-union and Yahoo graphs. Finally, as one odd anomaly, none of the reordering algorithms, including METIS, which has been experimentally tested for the better part of a decade, was able to a better ordering than the initial ordering. This leads us to suspect

that the graph-framework behind Twitter is doing some fairly interesting work on extracting locality from their massive graph.

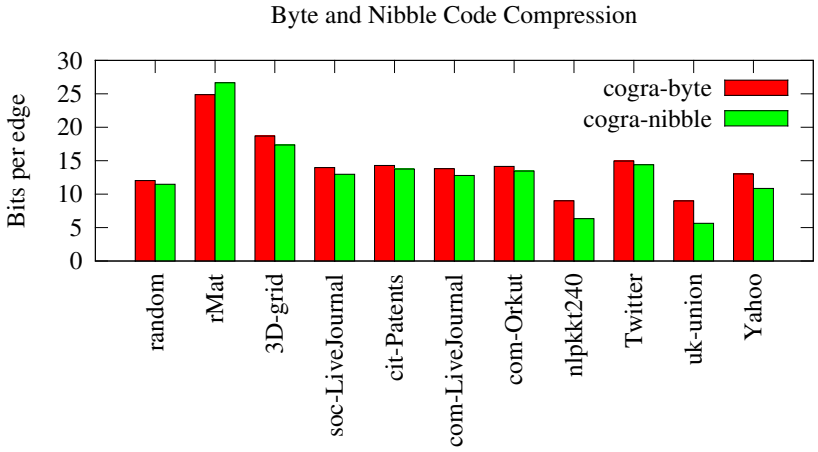


Figure 3.5: Number of bits per edge required for byte versus nibble coding.

Input Graph	gap	log	gap	log	gap	log	gap	log	gap	log	gap	log	gap	log	gap	log
Ordering	orig.	orig.	rand.	rand.	p-sep	p-sep	dfs	dfs	bfs	bfs	hybrid	hybrid	bfs-r	bfs-r	metis	metis
random	6.88	6.74	–	–	–	–	–	–	–	–	–	–	–	–	–	–
rMat	18.12	19.06	–	–	–	–	–	–	–	–	–	–	–	–	–	–
3D-grid	10.6	8.12	–	–	–	–	–	–	–	–	–	–	–	–	–	–
soc-LiveJournal	10.6	16.97	15.71	20.05	8.08	12.18	9.86	16.16	10.67	16.96	9.64	15.3	10.36	16.48	9.39	15.2
cit-Patents	16.43	19.48	17.97	20.35	8.57	10.1	11.7	16.37	12.3	17.53	11.66	15.09	13.0	16.39	10.25	13.98
com-LiveJournal	10.28	16.13	15.65	19.78	7.95	11.84	9.71	15.83	10.84	16.93	9.52	14.91	10.34	16.19	9.33	14.93
com-Orkut	10.42	17.5	13.61	19.39	8.58	14.53	10.09	17.7	10.35	17.85	9.87	17.26	10.16	17.74	10.03	16.85
nlpkkt240	4.49	23.74	19.28	22.57	4.13	8.18	5.1	14.27	4.02	17.44	3.81	11.17	3.15	8.56	3.87	10.61
Twitter	9.23	18.76	15.22	23.14	12.12	20.64	12.16	22.17	10.6	22.15	11.59	21.69	10.74	21.01	11.01	20.97
uk-union	3.14	11.44	17.08	24.83	–	–	3.0	13.39	3.01	18.62	2.31	14.41	–	–	–	–
Yahoo	7.6	24.56	21.33	28.22	–	–	6.56	18.09	7.14	23.34	6.22	17.66	–	–	–	–

Table 3.2: Average log cost and average gap cost of graph inputs using various ordering algorithms.

Input Graph	byte	nibble	byte	nibble	byte	nibble	byte	nibble	byte	nibble	byte	nibble	byte	nibble	byte	nibble
Ordering	orig.	orig.	rand.	rand.	p-sep	p-sep	dfs	dfs	bfs	bfs	hybrid	hybrid	bfs-r	bfs-r	metis	metis
random	12.03	11.48	–	–	–	–	–	–	–	–	–	–	–	–	–	–
rMat	24.88	26.65	–	–	–	–	–	–	–	–	–	–	–	–	–	–
3D-grid	18.68	17.34	–	–	–	–	–	–	–	–	–	–	–	–	–	–
soc-LiveJournal	16.76	16.37	22.26	23.12	**13.96**	12.98	15.93	15.36	16.89	16.49	15.8	15.09	16.55	16.06	15.18	14.75
cit-Patents	23.15	24.28	24.44	26.31	**14.29**	13.75	18.02	18.0	18.7	18.92	18.01	17.98	19.28	19.73	15.86	16.06
com-LiveJournal	16.4	15.96	22.24	23.06	**13.81**	12.8	15.74	15.16	17.09	16.72	15.64	14.93	16.49	16.03	15.06	14.93
com-Orkut	16.04	15.93	19.69	20.2	**14.14**	13.46	15.91	15.49	16.06	15.83	15.66	15.2	15.86	15.6	15.36	15.4
nlpkkt240	11.87	8.62	25.12	27.8	9.59	7.4	11.52	9.11	10.36	7.49	10.06	7.47	**9.0**	6.34	9.39	7.33
Twitter	**14.94**	14.4	21.4	22.38	17.98	18.24	18.15	18.3	16.58	16.24	17.53	17.56	16.51	16.38	16.75	16.75
uk-union	9.49	6.62	23.54	24.84	–	–	9.37	6.43	9.77	6.57	**8.98**	5.64	–	–	–	–
Yahoo	14.3	12.64	28.42	30.64	–	–	13.31	11.25	14.04	12.05	**13.02**	10.86	–	–	–	–

Table 3.3: Average bits per edge using byte codes and nibble codes. Storage for vertices is not included.

We also investigated the actual compression ratios of the reordering schemes, measured in the number of bits-per-edge. Tests were run using Cogra equipped with both nibble and byte

codes, and the resulting bits-per-edge values are charted in Table 3.3. We do not include the space required to store vertices, or vertex-degrees, as both the vertices and vertex-degrees are uncompressed in our framework. Furthermore, notice the direct correspondence between the reordering algorithm that achieves the minimum bits-per-edge for a given graph, and the reordering algorithm that minimizes the average log gap cost.

Figure 3.6 graphically depicts the average bits-per-edge required for compressing a graph using the best reordering algorithm for that graph on *byte*-codes. Notice that compression using nibble-codes almost always decreases the number of bits-per-edge, with the one exception of the rMat graph, which surprisingly requires more bits-per-edge to represent using nibble codes. This is because in rare cases when most values being compressed require ≈ 7 bits, byte-codes will be able to encode the values in a single byte, while nibble-codes will need to use 12 bits in order to store them. We point the reader to Figure 2.3 where this this situation is illustrated graphically.

Input Graph	Ligra	Cogra (byte)	Cogra (nibble)
random	433 MB	228 MB	221 MB
rMat	465 MB	444 MB	465 MB
3D-grid	278 MB	219 MB	209 MB
soc-LiveJournal	362 MB	188 MB	178 MB
cit-Patents	156 MB	107 MB	105 MB
com-LiveJournal	294 MB	152 MB	143 MB
com-Orkut	950 MB	440 MB	421 MB
nlpkkt240	3.1 GB	1.06 GB	815 MB
Twitter	12.08 GB	6.17 GB	5.95 GB
uk-union	45.9 GB	15.5 GB	10.9 GB
Yahoo	62.8 GB	37.9 GB	34.4 GB

Table 3.4: Total graph storage sizes, including both vertices and edges.

We also list the sizes required to store each graph in both Ligra, Cogra (byte) and Cogra (nibble) in Table 3.4, once again using the ordering that produced the best results from Table 3.3. This size includes both the edges, vertex offsets, and the vertex degrees. We note that while Ligra can implicitly store the degree of a vertex using the vertex offsets array, Cogra must explicitly store the degrees of each vertex. This leads to an extra $O(|V|)$ amount of space, which makes graphs that have large vertex/edge ratios appear to compress poorly in our framework. However, graphs where the number of edges is an order of magnitude larger than the number of vertices, such as nlpkkt240 or uk-union display a significant amount of space-savings compared to Ligra.

3.5.2 Performance and Memory Utilization

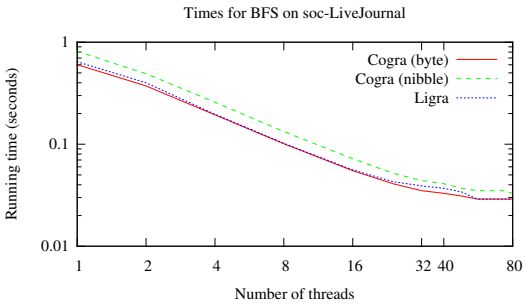
We now consider the performance of Cogra with respect to Ligra on the five applications described in Section 3.4. We do not include the time required to compress the graph in our timings, as this process happens only once, and is subsequently saved to disk. We denote the byte-encoded version of Cogra as Cogra (byte), and the nibble-encoded version of Cogra as Cogra(nibble). We report running times in a tabular format in Table 3.5. Each time is the median of three runs of the application.

Input Graph	Breadth-first Search			Betweenness Centrality			Graph Radii			Connected Components			PageRank		
	(L)	(C-b)	(C-n)	(L)	(C-b)	(C-n)	(L)	(C-b)	(C-n)	(L)	(C-b)	(C-n)	(L)	(C-b)	(C-n)
random	**0.056**	**0.056**	0.08	**0.151**	0.159	0.219	**0.289**	0.304	0.445	**0.0762**	0.0795	0.117	0.064	**0.0595**	0.081
rMat	**0.09**	0.092	0.121	**0.314**	0.341	0.481	**0.819**	0.898	1.33	**0.244**	0.276	0.426	0.219	**0.212**	0.295
3D-grid	0.219	**0.212**	0.234	0.574	**0.56**	0.605	**5.57**	6.08	8.28	**0.66**	0.703	1.41	0.041	**0.037**	0.045
soc-LiveJournal	0.029	**0.028**	0.033	**0.094**	0.096	0.136	**0.249**	0.256	0.405	0.09	**0.076**	0.135	0.062	**0.057**	0.08
cit-Patents	**0.031**	**0.031**	0.036	**0.086**	**0.086**	0.11	**0.191**	0.207	0.29	**0.047**	0.048	0.07	0.036	**0.033**	0.042
com-LiveJournal	**0.025**	**0.025**	0.029	**0.08**	0.084	0.117	0.189	**0.188**	0.305	**0.062**	0.067	0.112	0.048	**0.045**	0.062
com-Orkut	**0.03**	0.032	0.049	**0.139**	0.15	0.27	0.395	**0.379**	0.665	0.131	**0.107**	0.226	0.163	**0.14**	0.232
nlpkkt240	0.831	**0.463**	0.526	2.4	**1.36**	1.6	**22.3**	22.8	40.4	0.795	**0.589**	0.931	0.351	**0.222**	0.257
Twitter	**0.268**	0.28	0.352	4.65	**4.24**	6.58	7.5	**5.89**	7.76	3.25	**2.4**	3.84	2.45	**2.02**	3.03
uk-union	2.12	**1.44**	1.96	5.39	**4.0**	5.57	36.2	**16.8**	25.0	6.45	**2.73**	4.03	6.28	**2.56**	2.9
Yahoo	6.01	**3.87**	4.85	16.1	**13.1**	18.6	25.5	**23.5**	35.5	14.4	**10.1**	15.7	10.0	**7.47**	9.86

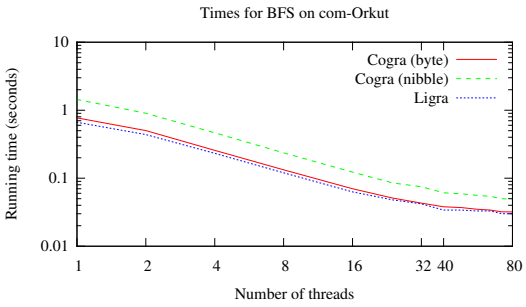
Table 3.5: Running times on 40 cores with hyper-threading on different applications for Ligra (L), Cogra using byte coding (C-b) and Cogra using nibble coding (C-n).

We also include speedup plots of the running time of Ligra, Cogra (byte) and Cogra (nibble) against the number of threads in Figure 3.6. The running time of Cogra (nibble) is almost always slower then the running time of Cogra (byte). The notable exception is for PageRank on nlpkkt240, where the nibble-encoded implementation nearly beats the byte-encoded implementation. We suspect this is due to the nibble-encoded representation of nlpkkt240 requiring ≈ 6 bytes-per-edge, whereas the byte-encoded representation requires ≈ 9 bytes-per-edge. The fact that the graph is more compressed in memory results in a reduction in the number of cache misses, which is likely what is responsible for the performance improvement.

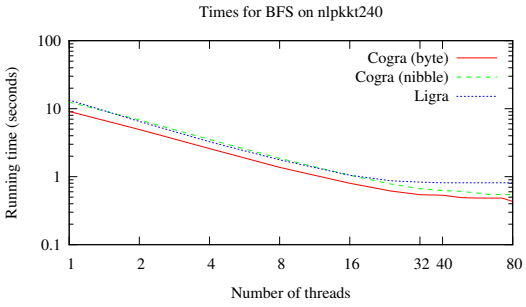
Lastly, we also plot the peak-memory usage for both Ligra, Cogra (byte) and Cogra (nibble) for several input graphs. We obtained memory usage information using Valgrind [35], using the *massif* tool. We then generated plots of the massif data using *ms_print*, and plotted the data for all three frameworks. Massif allows us to visualize the peak-memory usage for a number of snapshots taken during the applications life-time. We only profile how much heap-memory the programs use. We note that Cogra always have lower peak-memory usage than Ligra. For some graphs, however, Cogra uses memory on the same order as Ligra. However, this only occurs on graphs with a high vertex-to-edge ratio (the number of vertices and edges are roughly on the same order), when running applications that store auxiliary data-structures with size proportional to the number of vertices. For all other graphs, with low vertex-to-edge ratios, we observe a significant reduction in peak-memory usage compared with Ligra.



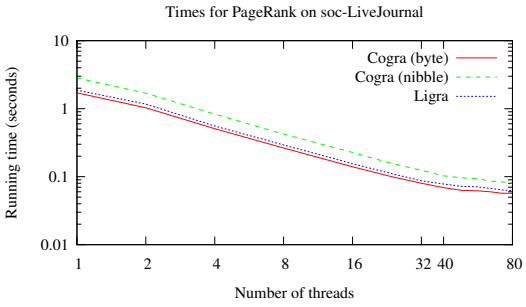
(a) BFS on soc-LiveJournal



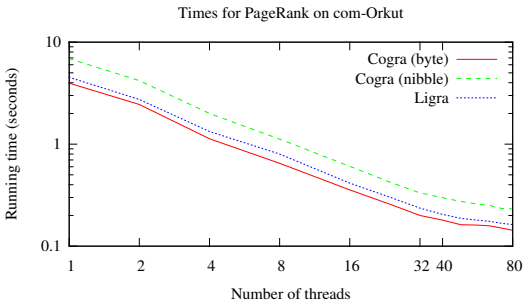
(b) BFS on com-Orkut



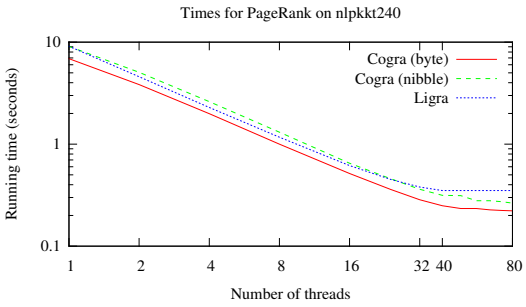
(c) BFS on nlpkkt240



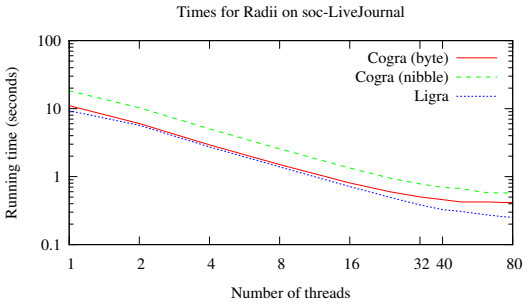
(d) PageRank on soc-LiveJournal



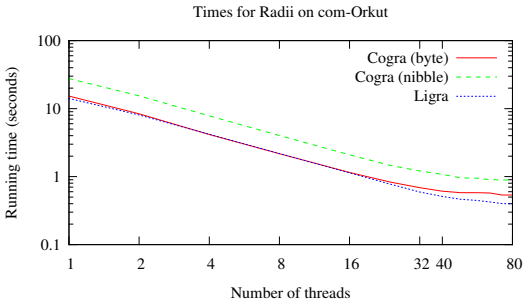
(e) PageRank on com-Orkut



(f) PageRank on nlpkkt240

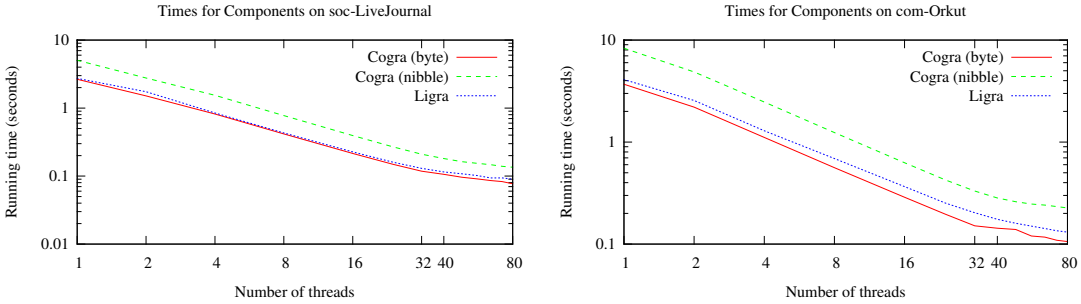


(g) Radii on soc-LiveJournal



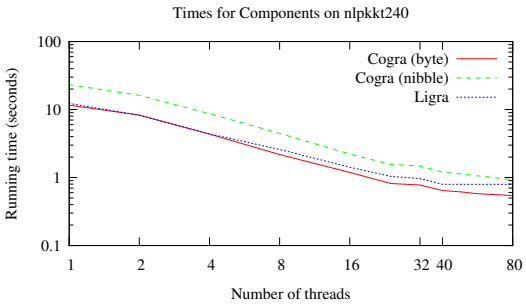
(h) Radii on com-Orkut

Figure 3.6: Times versus number of threads on various input graphs on a 40-core machine with hyper-threading. (40h) indicates 80 hyper-threads.

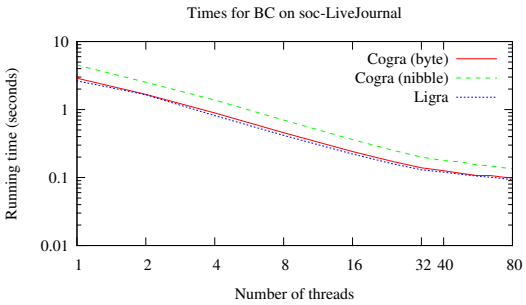


(a) Components on soc-LiveJournal

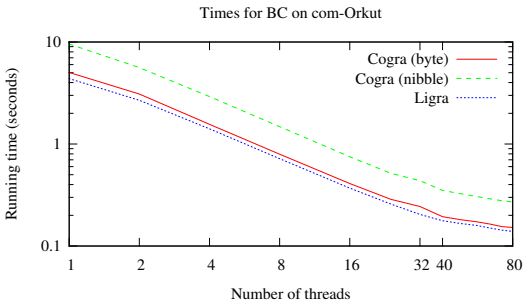
(b) Components on com-Orkut



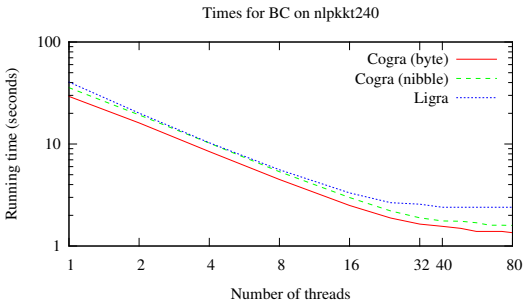
(c) Components on nlpkkt240



(d) BC on soc-LiveJournal

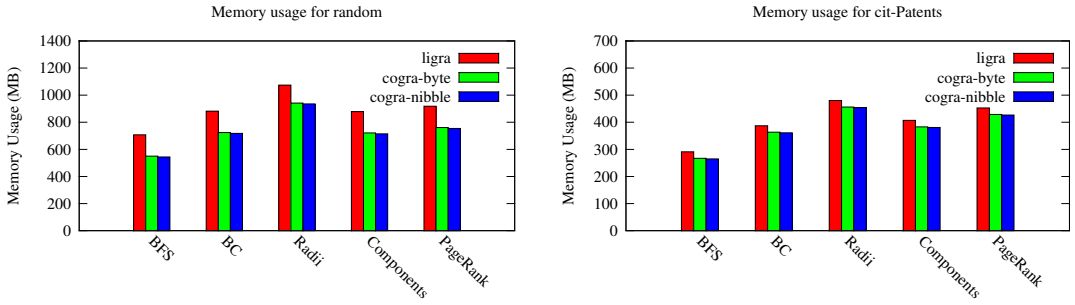


(e) BC on com-Orkut



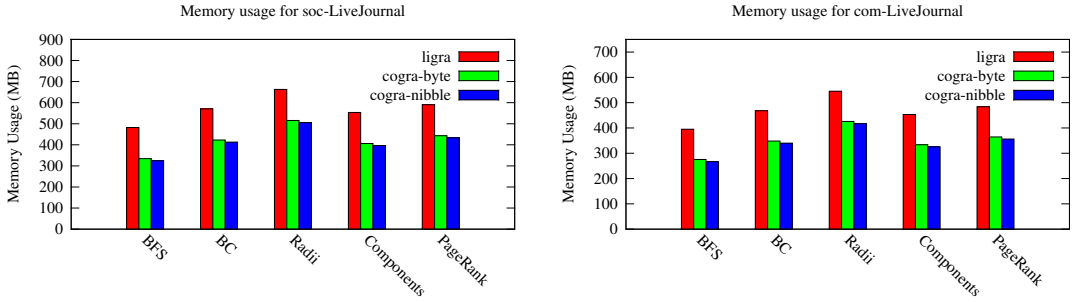
(f) BC on nlpkkt240

Figure 3.7: Times versus number of threads on various input graphs on a 40-core machine with hyper-threading. (40h) indicates 80 hyper-threads.



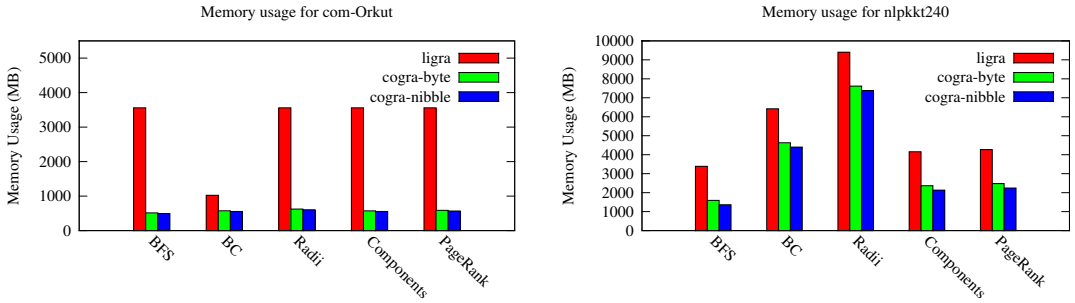
(a) random

(b) cit-Patents



(c) soc-LiveJournal

(d) com-LiveJournal



(e) com-Orkut

(f) nlpkkt240

Figure 3.8: Peak memory usage of graph algorithms on several inputs.

Chapter 4

Connectivity Labeling

We now turn our attention to the connectivity labeling problem, which illustrates the second approach to making graph algorithms highly performant, namely that of exploiting parallelism while ensuring that the resulting algorithm is work-efficient. In this chapter, we formally define the connectivity labeling problem, the difficulties found in attaining work-efficient algorithms, and review a number of historical as well as recent approaches to connectivity.

The connectivity labeling problem is a natural and basic problem on graphs that arises in a wide variety of fields such as VLSI design, and computer vision. The problem takes as input an undirected graph $G = (V, E)$ and assigns vertices within the same connected component identical labels, and vertices within different components unique labels. Two vertices, $u, v \in V$ are defined to be within the same component if there exists a path from u to v.

Sequentially, graph connectivity has a number of very natural solutions. Perhaps the simplest and most intuitive solution is to perform a breadth-first search starting at an arbitrary vertex. All vertices reachable from the initial vertex are given the same label. If any vertices remain once the search terminates, the algorithm increments its label and performs another breadth-first search starting at one of the unvisited vertices. An identical algorithm is possible using depth-first search.

Both the breadth-first and depth-first connectivity labeling require linear work. Union-find can also be effectively used in order to compute connectivity, and require nearly-linear work. However, parallelizing these algorithms is difficult. While breadth-first search can be run in parallel using techniques described in [3], depth-first search and union-find remain difficult to parallelize. We further note that any algorithm for connectivity must perform at least linear-work in the number of edges and vertices, as in the worst case, every edge will have to explored, and we return output proportional to the number of vertices.

4.1 Historical Approaches

Connectivity was one of the first problems to be extensively studied by the parallel algorithm community, and has a long history in the literature. The primary approach used in a number of these algorithms has been to compute a spanning-forest of the graph (possibly minimal) in parallel. To this end, a number of recent algorithms for computing minimum spanning forests in

parallel directly extend to create parallel connectivity algorithms. We first review a number of important results and algorithms on graph connectivity, covering spanning-forest methods, the Random Mate algorithm, and finally describe recent work on low-diameter decompositions.

4.1.1 Parallel Spanning-Forests

One of the first algorithms expounding this idea was published by Shiloach and Vishkin[50] and operates by applying fairly sophisticated tree-building and pointer-jumping techniques. Their technique involves two operations - 'hooking', and 'short-cutting', which hook one tree onto another, and decreases the height of a tree respectively. In addition to the original graph, G, they also maintain a pointer graph, which mutates over the course of the algorithm. At the end of the algorithm, every vertex has a pointer to the root of the tree, so querying whether two-vertices belong to the same tree can be done in constant time. Furthermore, a tree grows until it includes every vertex in a connected component, so two vertices can determine in constant time whether they belong to the same component. Extending this algorithm to the connectivity labeling problem is simple, as each vertex simply takes as its label the id of the root node of its tree.

We note that shortcutting is effectively a pointer-jumping technique, which sets every nodes pointer to the node its parent points to. By performing this operation often enough, deep trees are flattened into stars, which in turn increases the efficiency of hooking, and future short-cuts. As eventually, every vertex in a tree points to the root (the pointer graph becomes a collection of stars), the entire algorithm can be effectively thought of as hooks, with the added operation of pointer-jumping. The real subtlety in these algorithms involves strategies for hooking and short-cutting.

Shiloach and Vishkin [50] describe two methods, conditional-hooking, and unconditional-hooking. Conditional hooking melds two trees together, placing the root with a larger id as a child of the root with a smaller id. Unconditional-hooking is used when a tree has not been short-cut or hooked for one round. In this case, all vertices pointing to the stagnant root inspect their neighbors to see if one points to a different tree. In this case, they abandon the stagnant tree, and move themselves, and the subtree to the new tree.

Another algorithm later developed by Awerbuch and Shiloach [2] builds on the work of Shiloach and Vishkin, but modifies the unconditional hooking step. Instead of allowing any tree to unconditionally hook itself away from a stagnant root, their algorithm only allows stars to be unconditionally hooked. They focus on stars because membership queries within a tree are $O(1)$, which is desirable when testing whether two vertices come from the same tree.

Unfortunately, while both algorithms provided valuable insight on parallelizing connectivity, both the Shiloach and Vishkin algorithm, and the Awerbuch and Shiloach have super-linear work. While they both guarantee that the number of trees decrease by a constant fraction, and that we stop hooking trees in $O(\log n)$ rounds, they do not prove a bound on the number of edges removed per-round. Therefore, in the worst case, both algorithms require $O(m \log n)$ work. However, these algorithms can be made to be work-efficient by using somewhat complicated sampling techniques.

We also describe a recent algorithm by Patwary et al. [37]. They present a multi-core algorithm for computing spanning forests (with computing the connected components as a particular

application). Their main contribution is effectively parallelizing the union-find algorithm, which creates a special critical section for the Union operation in union-find. They provide a method of entering this critical section, which uses the expected strategy of locks. They parallelize the algorithm by adding a lock for every vertex. To prevent deadlock, they have each process acquire necessary locks in a predetermined order. They also present a lock-free approach which they call 'verification'. The technique involves having each processor storing edges that caused Union operations to be performed. At the end of the algorithm, each processor simply checks to ensure that the endpoints of all Union edges are in the same component (this ensures correctness). If this is not true, the edge is marked for re-processing.

4.1.2 Random-Mate

A more broadly useful technique known as ***Random Mate*** can also be applied to build connectivity algorithms. Random mate algorithms can be viewed as randomized versions of spanning-forest algorithms, where we randomize over hooking. In one formulation of random mate, vertices are assigned to be either star-centers, or satellites (the analogy is useful for exposition), with satellites contracting into star-centers. Both classes are assigned with equal probability. Algorithms using this technique are described by Reif [46] and Phillips [40].

Although the original papers did not describe the algorithm in the context of contracting the graph between each round, they can be easily extended to do so. If vertices are contracted, then the final round of the random-mate algorithm will produce a set of isolated nodes, which each represent a connected component of the graph. By maintaining contraction information between rounds, one can re-expand the graph in order to write the component id of each vertex. Although random-mate allows one to easily prove that $1/4$ of the vertices are removed each round, it not possible to get such a bound on the number of edges remaining after a round. Therefore, as the number of rounds is $O(\log n)$ with high probability, and on each round, we must do $O(m)$ work, the total work of random mate is $O(m \log n)$.

4.1.3 Work-Efficient Algorithms

A number of work-efficient polylogarithmic-depth connectivity algorithms have been designed [16, 19, 22, 23, 39, 42]. However, these algorithms are based primarily on random edge sampling, or on theoretical linear-work minimum spanning forest algorithms. Algorithms are usually based on sampling and filtering edges, using fairly sophisticated techniques to prove the work-efficiency of their methods. The algorithms are also fairly complicated and unfortunately do not appear to be practical, as there are no implementations of these algorithms. We now turn our focus to some theoretical tools which will be of use when designing a practical work-efficient connectivity algorithm.

4.1.4 Other Approaches

We also wish to saw a few words about other (not work-optimal) approaches to connected components that are encountered in practice. The first is a technique known as label-propagation. In label-propagation, each vertex is initially assigned a component id equal to its vertex id. Then, in

each round, each vertex inspects all of its neighbors. If any of its neighbors have a component id smaller than its component id, it updates its component id to this new minimum. The algorithm continues to perform this iteration over all vertices until no vertices update themselves, and a fixed point is reached. A quick analysis of a naive implementation of this algorithm shows that we perform $O((m+n)d)$ work in $O(d)$ depth where d is the diameter of the graph, as in the worst case, the minimum label takes d steps to propagate across the graph, and on each round we do $O(m+n)$ work as we inspect all edges. In practice, graph processing frameworks tend to implement an asynchronous algorithm [28, 51].

Very recently, a new algorithm based on empirical properties of real-world graphs was proposed by Slota et al. They describe a 'Multistep' method, which we now summarize. The key observation they use to build their new algorithm is that the majority of large real-world graphs tend to have a single large connected component, and a large number of smaller connected components. Therefore, their Multistep method picks a vertex – with high probability it is part of the large connected component – and runs a simple BFS in order to consume this component. Then, for the remaining components, they apply the label propagation algorithm that they refer to as a 'Coloring' algorithm. As the remaining components in the graph are all extremely small, the label-propagation only runs for $\max_{C_i} O(\text{diam}_{C_i})$ many rounds, where C_i are the individual components. Their approach can be effectively summarized as running 1) A fast parallel connectivity algorithm until a large component is removed from the graph, and 2) running label-propagation on the remaining vertices in the graph.

4.2 Low-Diameter Decompositions

The connectivity algorithm which we present in the following chapter is theoretically grounded in recent work on generating low-diameter decompositions in parallel. Intuitively, a low-diameter decomposition of a graph, $G = (V, E)$ is a partition of V into subsets such that the number of edges with vertices in different components is minimized. Concretely, a (β, d)-decomposition of an undirected graph $G = (V, E)$ is a partition of V into subsets $V_1, \ldots, V_k$ such that (1) the shortest-path distance between any two vertices in V_i, using only vertices in V_i is at most d, and that the number of edges $(u, v) \in E$ s.t. $u \in V_i$ and $v \in V_k$, $i \neq k$ is at most βm.

Sequentially, a very simple algorithm can be used to compute a low-diameter decomposition. Starting at an arbitrary vertex, the algorithm performs a breadth-first search, building a piece until either the number of edges on the frontier is a $\beta \cdot E'$ where E' is all edges within the piece. The algorithm then stops, and recurses on the subgraph induced by removing all vertices in the piece. As each piece has at most a β fraction of edges the total number of inter-piece edges is at most βm. Similarly, one can prove a strong diameter bound of $O(\log(n)/\beta)$ [32].

Recent work on the problem of computing-low diameter decompositions has emerged due to applications to solving SDD-linear systems (symmetric diagonally dominant linear systems) in parallel [7]. The initial algorithm presented, however, was fairly complicated, and required $O(m \log^2 n)$ work in order to compute a decomposition of quality $(\beta, O(\frac{\log^4 n}{\beta}))$. The method relies on growing the balls described in the sequential algorithm in parallel, if certain conditions are met.

More recently, Miller, Peng and Xu [32] revisited the problem of computing low-diameter de-

compositions of graphs and gave a simple and elegant algorithm for computing a $(\beta, O(\log n/\beta))$ decomposition in $O(m)$ work, and $O(\frac{\log^2 n}{\beta})$ expected depth with high probability. Their method also builds on the parallel ball-growing idea introduced in the earlier paper [7], but allows all balls to grow in parallel.

Their algorithm relies on the concept of 'shifted distances' and 'random shifts'. Given a graph, $G = (V, E)$, each $v \in V$ draws a value δ_v, independently from an exponential distribution with mean $1/\beta$. This value is called the random shift. The shifted distance between vertices u and v, denoted $d_{-u}(u, v)$ is then $d(u, v) - \delta_u$. Notice that values with a large value of δ_u experience a shortened distance to all vertices in the graph – this intuition is key to understanding their algorithm.

The second phase of the algorithm is an assignment process, and occurs after each vertex has drawn a random shift. Each vertex, v is assigned to piece S_u, started by the vertex u, which minimizes the shifted-distance to the v. We can view this algorithm as performing a parallel breadth-first search over a graph where all vertices are initially asleep. On each round, vertices that are awake perform one step of a breadth-first search originating from them. If any of their neighbors are currently unvisited, they are then acquired by the frontier vertex, and added to its piece. Initially, the vertex $v = \arg\max_{v \in V} \delta_v$ is the only vertex that is awake. Then, at time t, we add all vertices u s.t. $\delta_v - \delta_u < t$ to the multiple-source BFS.

We describe an intuitive way of thinking about this algorithm, which turns the previously described random shift on its head. As before, every $v \in V$ draws a random shift from the exponential distribution with mean $(1/\beta)$. Now, we compute $\delta_{max} = \max_{v \in V} \delta_v$, and set the random shift for vertex v to be $\delta_{max} - \delta_v$. We perform a multiple-source breadth-first search, where on round i, all vertices with shift-value δ_v between $i-1 < \delta_v \leq i$ are awakened and placed on the new frontier. Once again, awake vertices try to acquire vertices on their frontier. We have skipped over one crucial detail, namely how conflicts are resolved when a sleeping vertex v is on the frontier of two awake vertices, u, w. In this case, v is deterministically assigned to the vertex minimizing its shifted-distance, that is the vertex with the smaller random shift.

We now provide short explanations of some key facts. For a more comprehensive and detailed account, refer to [32].

Theorem 4.2.1 *With high probability, for all vertex v, $\delta_v \leq O(\frac{\log n}{\beta})$.*

Proof Using the CDF of the exponential distribution, $F(x) = 1 - e^{-\beta x}$ for non-negative x, we have that the probability that $\delta_v \geq (k+1) \cdot \frac{\log n}{\beta}$ is $1 - F((k+1) \cdot \frac{\log n}{\beta}) = e^{-(k+1) \cdot \frac{\log n}{\beta}} \leq n^{-(k+1)}$. Finally, applying union bound, over all vertices, we have that this quantity is $\leq n^{-k}$.

The high probability bound tells us the probability of any vertex's shift value not being in $O(\log(n)/\beta)$ is extremely low. Using this fact, we have that with probability $1 - \frac{1}{n^{-k}}$, the random-shifts algorithm produces a decomposition with strong-diameter $O(\frac{\log n}{\beta})$ (as the failure probability is $\frac{1}{n^{-k}}$ for a failure constant, k.

Consider an edge, (x, y), spanning between two pieces, S_u and S_v, that is $x \in S_u$ and $y \in S_v$. In order to bound the total number of inter-component edges, we first consider an arbitrary edge, and try to bound the probability that its endpoints, x and y are acquired by different vertices. This discussion follows more or less identically to the presentation in [32], and is augmented with some intuition gained from conversations with Miller and Xu.

The edge (x, y) is an inter-component edge only if $x \in S_u$ and $y \in S_v$, $u \neq v$. Let the midpoint of the edge x, y be denoted w. First, consider the set of all shifted distances to w, that is $\forall v \in V, d(v, w) + \delta_v$, and sort vertices by this quantity. The vertex that acquires w its piece, by definition, minimizes this quantity. Now, in terms of starting time, this is just $\forall v \in V, d(v, w) + \delta_{max} - \delta_v$. Removing the δ_{max}, as it appears in all equations, we have that the vertex acquiring w for its piece is the vertex $v \in V$ that minimizes $d(v, w) - \delta_v$.

Now, consider the vertex w^*, which minimizes the distance to w. It is easy to see that $w*$ is one of $\{u, v\}$, as w is the midpoint of the edge (x, y), and $x \in S_u$, and $y \in S_v$. Given that w^* minimizes the distance to w, the paper now shows that

$$d_{-\delta}(u, w), d_{-\delta}(v, w) \leq 1 + d_{-\delta}(w^*, w)$$

or that the shifted distance from w to u or v is at most one more than the minimum shifted distance to w. Notice that $d_{-\delta}(w^*, x), d_{-\delta}(w^*, y) \leq 1/2 + d_{-\delta}(w^*, w)$, as w is the midpoint of the edge (x, y), and the distance from x, y to w is $1/2$. The proof follows by simple contradiction, assuming $d_{-\delta}(u, w) > 1 + d_{-\delta}(w^*, w)$. This argument gives us that if $x \in S_u$, $y \in S_v$, $u \neq v$, then the shifted distance $d_{-\delta}(u, x)$ and $d_{-\delta}(v, y)$ differ by at most one. Therefore, in order to bound the probability that an arbitrary edge $(u, v) \in E$ is cut, we must bound the probability that the minimum shifted distance to u, and the minimum shifted distance to v differ by less than 1, as if the difference is greater than 1, both u and v will belong to the same piece.

The final argument we present from [32] formalizes this notion to compute the probability that an arbitrary edge, $(u, v) \in E$ has $u \in S_u$, $v \in S_v$, $u \neq v$. As argued previously, an edge is cut only when the difference between the smallest and second smallest shifted distances is less than 1 (otherwise, both endpoints of the edge are included in the vertex which achieves the minimum shifted-distance). We now prove the following lemma regarding the probability of an edge being cut.

Lemma 4.2.2 *For an arbitrary edge $(x, y) \in E$, $x \in S_u$, and $y \in S_v$ with probability β.*

Proof By our previous argument, an edge goes between pieces exactly when the difference between $|d_{-\delta}(u, x) - d_{-\delta}(v, y)| < 1$. [32] present a very elegant analysis of this situation - they view each shifted distance as some arbitrary value d_i, in addition to a value sampled from $Exp(\beta)$. Using the analogy of light-bulbs, the d_i can be seen as some fixed times when a given light-bulb is turned on. Then, each light-bulb has lifetime distributed according to $Exp(\beta)$. We must now compute the probability that the time the last light-bulb goes off minus the time the second to last light-bulb goes off is less than 1. Using the memorylessness property of Exp, we have that this is simply the probability $P(X < 1)$ where $X \sim Exp(\beta)$. Using the CDF of the exponential, this is just $1 - e^{-\beta}$, which they approximate as $1 - (1 - \beta) = \beta$. Identical analysis allows us to conclude that $P(X < c) \approx c\beta$, which will be necessary for our use of this lemma.

4.3 Extending Low-Diameter Decompositions to Connectivity

Given a black-box algorithm for computing the low-diameter decomposition of a graph $G = (V, E)$, we can create a simple algorithm for computing the connectivity labeling of G. The algorithm first calls the black-box decomposition algorithm and obtains a low-diameter decomposition, which induces some k disjoint subsets of $V = \{V_1, \ldots, V_k\}$. We then contract each

V_i, placing a single node for each V_i in G', and relabeling edges that cross different pieces in the decomposition. We then recurse on this graph, until we are left with a collection of singleton nodes.

Using the low-diameter decomposition algorithm designed by Miller et al. [32] as our black-box algorithm during a single round guarantees that at most βm edges span different components in expectation. Therefore, with β set to a constant fraction, the contracted subgraph in the next round only has a constant fraction of the original edges. This implies that the number of rounds the algorithm runs is $O(\log_{\frac{1}{\beta}} m)$ in expectation. The connectivity algorithm described in the following chapter formalizes this intuition. The critical difference between this connectivity algorithm and previous contraction-based approaches to connectivity is that the number of edges decreases by a constant fraction each round, which allows us to prove the work-optimality of our technique.

In the final chapter, we formally present this new connectivity algorithm, describe two versions of the low-diameter decomposition algorithm and their theoretical guarantees, and finally detail the experimental results of the algorithm.

Chapter 5

Simple Work-Efficient Connectivity

We have described a number of techniques for computing the connectivity labeling of a graph, including computing spanning-forests, random-mate, and sampling-based techniques. Finally, we described the recent development of a fast parallel algorithm for computing low-diameter decompositions of a graph [32]. Using this algorithm as a sub-routine, we described a simple connectivity algorithm. In this chapter, we formally describe this connectivity algorithm, show the theoretical guarantees of a non-deterministic version of this algorithm and finally describe our implementation of the algorithm as well as its experimental performance.

Our main contributions in this work are the development of a simple linear-work, polylogarithmic depth parallel algorithm for connectivity, and a highly optimized implementation of this algorithm that is competitive with the fastest existing parallel implementations of graph-connectivity.

5.1 A Simple Algorithm

We will make extensive use of the notation and tools from Miller et al [32]. The reader is advised to refer to Section 4.2 for a short description of some of their main results, or the original paper for a full account. We now describe our first connectivity algorithm that uses a deterministic version of the low-diameter decomposition.

Algorithm 1 Parallel decomposition-based algorithm for connected components labeling

```
1:  β = some constant fraction in (0, 1)
2:  procedure CC(G(V, E))
3:      L = DECOMP(G(V, E), β)              ▷ L contains the labels returned by DECOMP
4:      G'(V', E') = CONTRACT(G(V, E), L)
5:      if |E'| = 0 then
6:          return L
7:      else
8:          L' = CC(G'(V', E'))
9:          L'' = RELABELUP(L, L')
10:         return L''
```

Algorithm 1 shows pseudo-code for our parallel connectivity labeling algorithm. It uses a low-diameter decomposition, labeled DECOMP as a sub-routine, which takes as input a graph $G = (V, E)$ and a value $\beta, 0 < \beta < 1$. Decomp implements the low-diameter decomposition algorithm described in Section 4.2, which runs in $O(\log^3 n)$ depth and $O(m)$ work in expectation. DECOMP then returns a labeling of $v \in V$ into $[0, k)$, where k is the number of pieces generated by the decomposition. The algorithm itself is very simple and easy to understand. We use DECOMP to compute a low-diameter decomposition of G. CONTRACT then contracts all vertices in a given piece to a single vertex, with only inter-piece edges remaining, and returns a graph G'. If G' only has singletons remaining, we return the labeling induced by DECOMP, as we have decomposed the original graph into its connected components. Otherwise, we recurse, computing the labeling for G', and then compute a labeling for G using the procedure RELABELUP. RELABELUP simply computes $L'[L[v]] \forall v$, which is necessary in order to propagate label information from the recursive call to CC.

5.1.1 Theoretical Guarantees

We now prove the expected work and depth of our algorithm.

Theorem 5.1.1 *Algorithm 1 runs in $O(m)$ work and $O(\log^3 n)$ depth in expectation on a CRCW PRAM.*

Proof We first consider the number of iterations this algorithm runs in expectation. As DECOMP returns a decomposition of V with at most βm inter-piece edges, the number of edges in G', the contracted graph, is at most βm, without the removal of any multi-edges. For β set to some constant, $0 < \beta < 1$, we observe a geometric decrease in the number of edges between each round. Therefore, the total number of recursive calls is $O(\log_{\frac{1}{\beta}} m)$ in expectation.

Furthermore, in each recursive call we have a call to DECOMP, which runs in $O(m')$ work and $O(\frac{\log^2(n)}{\beta})$ depth in expectation, where m' is the number of edges in this round. Assuming that we can perform CONTRACT and RELABELUP within the timebounds of DECOMP, that is $O(m)$ work and $O(\frac{\log^2(n)}{\beta})$ depth, CC runs in $O(m)$ work and $O(\log_{\frac{1}{\beta}} m \frac{\log^2 n}{\beta}) = O(\log^3 n)$ work ($\log m \in O(\log n)$).

We now describe how to implement DECOMP, CONTRACT and RELABELUP to meet these time-bounds. The simplest implementation of DECOMP simply runs a multiple breadth-first search algorithm in parallel, where vertices are staged to start at various times sampled from an exponential distribution. Each active BFS in the graph corresponds to a single piece produced by the decomposition. We can maintain this multiple BFS using a single frontier-array, where all vertices belonging to a single piece are placed consecutively in the frontier. Furthermore, as we place exactly n vertices on our frontier, we can pre-allocate this frontier array and maintain an offset into it that increases each round. Furthermore, recall that DECOMP produces a maximum-shift value of $O(\frac{\log n}{\beta})$, and therefore, we have $O(\frac{\log n}{\beta})$ frontiers in total with high probability. As each piece's vertices are consecutive, each piece simply maintains an index into each frontier where its vertices start and end.

Notice that after one iteration of BFS, each vertex processed on the current frontier can compact its own adjacency list to only include edges that will be included in the contracted

graph (the edge array only includes inter-piece edges, with multi-edges still remaining). This process involves simply maintaining an index while iterating over the vertex's adjacency list, and compacting the list by writing over edges going between two vertices in the same piece. Therefore, during the multi-source BFS, we can also maintain the number of edges for each vertex (by rewriting the vertex's degree), and perform both adjacency list compaction in $O(m')$ work and $O(\log m')$ depth where m' is the number of edges in the current graph. We now describe how to implement CONTRACT using this information produced by DECOMP.

First, notice that using the rewritten vertex adjacency lists, which only have inter-piece edges, we can compute a new array consisting of only inter-piece edges (by using a prefix-sum on the degree array, and in parallel copying each vertex's inter-piece edges into this new array). However, there may still be duplicate edges which we must remove. We can now remove these duplicate edges from the array using hashing, which can be done in $O(m')$ work and $O(\log m')$ depth [31]. We emphasize that the number of edges goes down by a constant factor even without removing duplicates (multi-edges) in G'. We will later describe an empirical observation that the number of edges decreases by another large factor when removing duplicates.

We now have an edge-array consisting of all inter-piece edges. However, the vertex ids of these edges are still in the original range, $[0, n)$, and in order to contract the graph we must relabel them to be in the range $[0, k)$ where k is the number of pieces generated by DECOMP. We do this once again by using a prefix sum to compute a map, L', from vertices to their ids in the contracted graph. This map is then used to relabel the edges endpoints to be in $[0, k)$. We then pass the correctly labeled graph into the recursive call of CC, and obtain a labeling L'' which maps the contracted vertices to their true connectivity labels which are in the range $[0, k')$.

L'', the result of the recursive call is then taken by RELABELUP, which simply maps the original vertices to their new connectivity labels, which can be easily done in $O(m')$ work and $O(\log m')$ depth, as for each $v \in V$ we simply compute $L''[L'[v]]$ in parallel. Therefore, we have shown that both CONTRACT, RELABELUP and DECOMP can be performed in $O(m')$ work and $O(\log^2 n')$ depth in expectation, giving us a total of $O(m)$ work and $O(\log^3 n)$ depth in expectation.

5.1.2 Allowing Non-Determinism

We now describe a small modification of the algorithm which makes the low-diameter decomposition non-deterministic, and describe the theoretical properties of the connectivity algorithm as a result of this non-determinism. First, notice that the determinism or non-determinism of our connectivity algorithm is dependent solely on the choice of a deterministic or non-deterministic low-diameter decomposition, as all other steps are deterministic given a choice of decomposition.

Recall that in the low-diameter decomposition algorithm described in 4.2, we determined which piece a vertex, u,was added to by computing the shifted distance for each $v \in V$, and choosing the vertex which achieved the minimum, using the fractional part of the shifted-distance in order to break ties. That is, u was assigned to the vertex v minimizing:

$$\arg\min_{v \in V} d(v, u) - \delta_v$$

As the exponential distribution has support on the non-negative reals, we had two vertices, x, y tying for ownership of a vertex u being a zero-probability event. Furthermore, the analysis which

bound the probability of an edge lying between two pieces as β was dependent on the fact that the edge's endpoints belong to different pieces only when the smallest, and second-smallest shifted-distances to the midpoint of the edge different by less than 1.

Now, suppose that instead of choosing the vertex achieving the minimum shifted-distance, we randomly choose one of the vertices that has shifted-distance < 1. We first claim that this modification is equivalent to the following two situations. Once again, consider each $v \in V$ drawing values independently from $Exp(\beta)$, but instead of keeping the fractional part of each value, we round down all values to the nearest integer. For a vertex, u, the vertex v that u is assigned to is still computed by calculating $\arg\min_{v \in V} d(v, u) - \delta_v$, but now ties are broken arbitrarily amongst vertices achieving this minimum.

A more natural algorithmic depiction of this situation is given as follows. Once again, we perform a multi-source BFS, starting vertices at a given round if their shift-value is less than the current time-step. Now, for some vertices u, v on the current frontier, where the edges (u, w) and (v, w) both exist to some asleep vertex w, we allow either u or v to acquire w for its piece. We now show that we only see twice as many inter-piece edges in expectation when using this random tie-breaking strategy.

Lemma 5.1.2 *Breaking ties arbitrarily in the low-diameter decomposition algorithm produces an $O(2\beta, O(\frac{\log n}{\beta})$ decomposition with high probability.*

Proof Firstly, in order to prove the strong diameter bounds, notice that we are still picking random-shifts from $Exp(\beta)$, and our previous proof showing that the value of the maximum shift is in $O(\frac{\log n}{\beta})$ with high probability still holds. This proves the bound on diameter, because after $\delta_{max} \in O(\frac{\log n}{\beta})$ rounds, the algorithm terminates as every vertex 'wakes up'. Secondly, we must show that the number of inter-piece edges is at most $2\beta m$.

Once again, consider w, the midpoint of an edge (x, y), where $x \in S_u$ and $y \in S_v$. Recall that if x and y are placed in different pieces, then $d_{-\delta}(u, w)$ and $d_{-\delta}(v, w)$ are within one of the minimum shifted distance to w. As the arbitrary argument just rounds down all shifted-distances to the nearest integer (it rounds the random-shifts down, which correspond to rounding down the shifted-distances as we operate on an unweighted, undirected graph). Now, two rounded shifted-distances differ by at most one when the original un-rounded distances differ by at most two.

Thus, by applying Lemma 4.2.2 which showed that the probability that the smallest, and second-smallest shifted-distances are c apart is $c\beta$ with $c = 2$, we have that this probability is at most 2β. Thus, the probability that an edge is cut, or has its endpoints in two separate pieces is at most 2β. By linearity of expectations, we have that the total number of cut edges is at most $2\beta m$, which concludes our proof.

As a corollary, for $0 < 1/2 < \beta$, we can plug in the arbitrary-rounding decomposition algorithm into Theorem 5.1.1 to obtain a linear-work, polylogarithmic depth connectivity algorithm. Continuing onwards, we refer to the arbitrary-rounding algorithm as DECOMPARB. We will give a more thoroguh treatment of DECOMPARB when describing implementation details of both DECOMP and DECOMPARB.

5.2 Implementation

We implemented two variants of our connectivity algorithm, CCDET and CCARB, which correspond to a deterministic and non-deterministic version respectively. Both algorithms are implemented in $C++$ using Cilk Plus to express parallelism. As the primary difference between CCDET and CCARB is in their choice of decomposition algorithm, we will focus on our implementation of the two decomposition algorithms, and describe various optimizations made.

Both algorithms require us to draw random-shfits from $Exp(\beta)$. Instead of doing this using $C++$'s STD::EXPONENTIAL_DISTRIBUTION, we opted use the suggestion provided by [32], and simulate the exponential distribution by first generating a random permutation of the vertices, and then simulating shifts by adding $e^{i\beta}$ many vertices on the i'th round (we add chunks of vertices taken from this random permutation in order, until all chunks have been used, which occurs for $i = \log(n)/\beta$.

We represent a graph using the adjacency array format. In this representation, we maintain two arrays. The first array consists of all edges in the graph, represented by their target endpoint. The second is an array of vertex offsets, where each vertex points to the start of its adjacency list in the edge array. Using the vertex offset array, we can implicitly store the degree of vertex i, which is recovered by computing offset$[i+1]$ − offset$[i]$. We set offset$[n-1] = m$. We also denote the current frontier as $\mathcal{F}$, and the next frontier as $\mathcal{F}'$.

We first give a general account of techniques used in both algorithms. Because our simulation of the exponential distribution consists of generating a random permutation and choosing exponentially larger chunks from this permutation to add onto the next frontier, we have no access to a fractional shift-value when breaking ties. To remedy this, we also draw integers at random from a large-enough range that w.h.p. there are no ties, and use these numbers to simulate the fractional parts of the shift-value. We denote these integers representing the fractional parts of the shift value by δ'. The algorithm proceeds as a multiple-source BFS, and stores each frontier in a single array of size $|V|$. Because we visit exactly n vertices, we can pre-allocate this space, and write each successive frontier into the array in parallel.

In the deterministic algorithm, DECOMP we have to deterministically guarantee that only the vertex with the smallest shifted-distance to a target vertex v is allowed to mark v with its piece's id. In order to implement this, we create an array C which stores a single pair for each vertex. Initially, all pairs are initialized to (∞, ∞). For a vertex v, the first component of C, denoted $C_1[v]$, is used to resolve conflicts when multiple vertices on a frontier are trying to acquire v. The second component of $C[v]$, denoted $C_2[v]$, is used to mark the vertex id of the piece that v is assigned to.

The algorithm runs a two-phase BFS. When vertices are being added as new piece centers, we set $C[v] = (-1, v)$, where -1 indicates that this vertex has already been visited, and the v indicates that the piece id is its own vertex id. In the first phase, all vertices on the frontier iterate over their out-edges. Then, for each edge (u, v), where $u \in \mathcal{F}$ and v is an unvisited vertex, u uses a writeMin operation and writes a pair, (u_{id}, u_{frac}) where u_{id} denotes the id of u's piece, and u_{frac} denotes the integer representing the fractional part of u's shift-value. We use the writeMin in order to ensure that the vertex with the smallest fractional shift-value acquires the target vertex out of all other vertices on the current frontier competing for the target.

DECOMP then carries out a second BFS, where for each $u \in \mathcal{F}$, u examines its out-neighbors

Algorithm 2 Decomp

```
1: C = {(∞,∞),...,(∞,∞)}
2: Frontier = {}
3: numVisited = 0
4: while (numVisited < n) do
5:     add to Frontier unvisited vertices v with δ_v < round + 1
6:       and set C[v] = (−1, v)                                  ▷ new BFS centers
7:     numVisited = numVisited + size(Frontier)
8:     NextFrontier = {}
9:     parfor v ∈ Frontier do
10:        start = V[v]                                          ▷ start index of edges in E
11:        k = 0
12:        for i = 0 to D[v] − 1 do
13:            w = E[start + i]
14:            if C_1[w] ≠ −1 then
15:                if C_1[w] > δ'_{C_2[v]} then
16:                    writeMin(C[w],(δ'_{C_2[v]}, C_2[v]))
17:                E[start + k] = w
18:                k = k + 1
19:            else
20:                if C_2[w] ≠ C_2[v] then
21:                    E[start + k] = −C_2[w] − 1
22:                    k = k + 1
23:        D[v] = k
24:    parfor v ∈ Frontier do
25:        start = V[v]                                          ▷ start index of edges in E
26:        k = 0
27:        for i = 0 to D[v] − 1 do
28:            w = E[start + i]
29:            if w ≥ 0 then
30:                if C_1[w] = δ'_{C_2[v]} and CAS(C_1[w], δ'_{C_2[v]}, −1) then
31:                    add w to NextFrontier                     ▷ v won on w
32:                else
33:                    if C_2[w] ≠ C_2[v] then
34:                        E[start + k] = −C_2[w] − 1
35:                        k = k + 1
36:            else
37:                E[start + k] = w
38:                k = k + 1
39:        D[v] = k
40:    NextFrontier = Frontier
```

once more. For each unvisited vertex v that u visits, u checks the value written to v. If $C[v] == C[u]$ (both the fractional value, and the component id are the same), then u uses a compare-and-

swap to atomically acquire v. The compare-and-swap is necessary for correctness to ensure that v is only added a single time to the next frontier.

Examining Algorithm 2, notice that we also perform rewriting, or 'packing out' of a vertex's adjacency list while performing the BFS. In the first phase of the BFS, we first check to see whether the vertex in $\mathcal{F}'$ has already been visited. If it has not, we perform the usual writeMin operation, trying to place a reservation for the vertex, and preserve the edge until the second BFS (lines 17 and 18). Otherwise, if the vertex has already been visited, we check to see whether the id of the piece it belongs to is different from our own. If this is the case, we preserve the edge (lines 21 and 22). Otherwise, the edge is written over, as it goes between two vertices in the same component. Finally, we set our degree to be k, as only the first k edges in our adjacency list are relevant.

We also perform rewriting in the second BFS. In the second phase, we have eliminated all intra-piece edges for vertices that were acquired in previous rounds. We may however still have some intra-component edges that are created by vertices added in this round, and must therefore eliminate these. Just as in phase 1, we maintain a variable k used to compute our new degree. If we win the compare-and-swap for a vertex on our frontier, then the edge (u, v) is intra-component and is not preserved, and is not included in the degree of u. Otherwise, if we lose the compare-and-swap, we must check the piece-id of the vertex. If it is different than our own, we must rewrite the edge, and increment k. Therefore, at the end of the second BFS, the first $Deg[v] = k$ vertices in our adjacency list are solely inter-piece edges.

Algorithm 3 Decomp-Arb

```
 1: C = {∞, ..., ∞}
 2: Frontier = {}
 3: numVisited = 0
 4: while (numVisited < n) do
 5:     add to Frontier unvisited vertices v with δ_v < round + 1
 6:       and set C[v] = v                                  ▷ new BFS centers
 7:     numVisited = numVisited + size(Frontier)
 8:     NextFrontier = {}
 9:     parfor v ∈ Frontier do
10:         start = V[v]                                    ▷ start index of edges in E
11:         k = 0
12:         for i = 0 to D[v] − 1 do
13:             w = E[start + i]
14:             if C[w] = ∞ and CAS(C[w], ∞, C[v]) then
15:                 add w to NextFrontier
16:             else
17:                 if C[w] ≠ C[v] then                     ▷ inter-component edge
18:                     E[start + k] = C[w]
19:                     k = k + 1
20:         D[v] = k
21:     NextFrontier = Frontier
```

We now describe the implementation of our second low-diameter decomposition algorithm,

CCARB. The second connectivity algorithm, CCARB is non-deterministic solely because it uses DECOMPARB as its low-diameter decomposition subroutine instead of DECOMP. We first implemented DECOMPARB, and toyed with non-determinism due to the fact that DECOMP was fairly slow due to the fact that every edge was inspected twice in two BFS phases per frontier. Therefore, we investigated replacing the two-phase BFS with just a single BFS phase, where all vertices on the current frontier simply used compare-and-swap to acquire unvisited vertices.

To this end, consider the non-deterministic low-diameter decomposition algorithm described in Algorithm 3. The operation of the algorithm is similar to Algorithm 2 except for the fact that we only perform a single BFS. We once again use rewriting to 'pack out' unnecessary edges and only preserve inter-piece edges that will become a part of the contracted graph. Furthermore, instead of using a writeMin in order to reserve vertices, each vertex simply uses a compare-and-swap in order to acquire a vertex on the new frontier atomically. This ensures that vertices appear exactly once on a frontier, and are not duplicated. The vertex in the current frontier that wins a compare-and-swap is then responsible for adding the acquired vertex to the next frontier.

Because DECOMPARB only makes a single pass over the out-edges of each frontier (instead of two passes as in DECOMP), we expected DECOMPARB to have better overall performance and speedup compared with DECOMP. As we will shortly see when describing our experiments, there is a significant difference between DECOMP and DECOMPARB, indicating that determinism in algorithms often comes at a price. We wish to comment that one can see similar effects in the graph-processing systems mentioned in Chapters 1 and 2 of this work, where systems that used BSP models were often orders of magnitude slower than systems that allowed for asynchronous updates.

We also implemented a third version of the decomposition algorithm, called DECOMPARB-HYBRID, which uses the direction optimizing breadth-first search introduced by Beamer et al. [3]. We gave a thorough overview of the direction optimizing BFS in Section 3.2.1, and its applicability to Ligra and Cogra. We now consider the direction-optimizing breadth first search and its applicability to our connectivity algorithm.

Notice that even if we alternate between the 'bottom up' and 'top down' breadth-first search, we must still inspect all edges from a vertex v on the current frontier. This is because we actually compact v's adjacency array while doing the breadth-first search in order to contract the graph within the time bounds. Therefore, if we apply the optimization, and have all unvisited vertices in the graph inspect their in-edges when the frontier is dense, we must apply a post-processing step where each vertex goes through its edges and compacts out intra-piece edges.

5.3 Experiments

We have presented a theoretical algorithm for computing the connectivity labeling of a graph and described three implementations of the algorithm, DECOMP, DECOMPARB, and DECOMPARB-HYBRID. We now investigate the experimental properties of these algorithms, and benchmark them against a number of other algorithms for connectivity.

Our experimental setup is covered in Section 1.1. We once again make use of the ***rMat***, ***randLocal*** and ***3D-grid*** graphs described previously. We introduce a second version of the rMat graph, called ***rMat2***, which uses the same generator as rMat, but has a higher edge-to-vertex

ratio, which results in a denser graph. We also introduce the ***line*** graph, which as its name suggests is a path of length $n-1$. The graph has diameter $n-1$, and is a degenerate edge case to measure how our connectivity algorithms perform on high-diameter graphs. We also use the *com-Orkut* graph, which is a social network graph downloaded from the Stanford Network Analysis Project (SNAP) [1]. We note that the generated graphs are not represented in the 'local' order described earlier, but are given in a random order for consistency. Table 5.1 describes the sizes of the graphs used in our experiments.

Input Graph	Num. Vertices	Num. Edges
random	10^8	5×10^8
rMat	2^{27}	5×10^8
rMat2	2^{20}	4.2×10^8
3D-grid	10^8	3×10^8
line	5×10^8	5×10^8
com-Orkut	3,072,627	117,185,083

Table 5.1: Input graphs

We compare our algorithms to several publicly available parallel implementations of connectivity. To the best of our knowledge, these are the fastest available parallel connectivity algorithms that we are aware of [37, 52]. We refer to the algorithm using DECOMP as ***decomp-min-CC***, the algorithm using DECOMPARB as ***decomp-arb-CC***, and lastly the algorithm using DECOMPARBHYBRID as ***decomp-arb-hybrid-CC***.

The first algorithm we benchmark against is by Patwary et al. [37]. The implement two versions of their algorithm, one using locks, and the other using verification. We were successful in running their lock-based implementation, but found that their verification-based algorithm sometimes failed to terminate. Both algorithms are based off of union-find. We also noticed that their lock-based algorithm typically outperformed the verification-based algorithm. We refer to the lock-based algorithm as ***parlalel-SF-PRM*** in our experiments. We also compare our algorithm against the parallel spanning forest implementation of connectivity that is provided in the Problem Based Benchmark Suite [52]. We refer to this algorithm as ***parallel-SF-PBBS***. Both of these parallel implementations of connectivity have been found to work well in practice, but unfortunately neither are theoretically work-efficient.

We also include a baseline serial algorithm, which implements a simple union-find algorithm, which we refer to as ***serial-SF***. This algorithm can be found in the PBBS, along with the parallel spanning forest algorithm. We found that breadth-first search was slower than the union find algorithm (mostly due to BFS's difficulty with high diameter graphs) and therefore use the union-find algorithm as our serial benchmark.

We report running times for all algorithms in Table 5.2. For each graph, we report the serial running time, as well as the 40h time, which is the running time on 40 cores with hyper-threading enabled. We report the median of three trials for each time. We observed that decomp-arb-CC and decomp-arb-hybrid-CC almost always outperform decomp-min-CC, which is likely due to the fact that decomp-min-CC must pass over each edge in the frontier twice, whereas decomp-arb-CC and decomp-arb-hybrid-CC only require a single pass over the edges in the frontier.

Implementation	random		rMat		rMat2		3D-grid		line		com-Orkut	
	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)	(1)	(40h)
serial-SF	19.5	–	21.5	–	2.86*	–	17.5	–	68.6	–	0.82*	–
decomp-arb-CC	43.1	1.97	46.7	2.5	6.95	0.256	30.1	1.36	254	6.49	2.35	0.115
decomp-arb-hybrid-CC	38.7	1.89	39.8	2.22	4.11	0.116	30.6	1.39	247	6.5	1.22	0.058
decomp-min-CC	74.8	2.86	76.3	3.49	7.22	0.221	57.9	2.11	348	9.11	2.39	0.132
parallel-SF-PBBS	70.9	1.91	79.2	2.13	9.79	0.515	41.1	1.53	174	5.22	2.98	0.156
parallel-SF-PRM	48.8	1.64	42.2	1.3	4.51	0.1	30.3	1.33	313	4.02	1.25	0.04

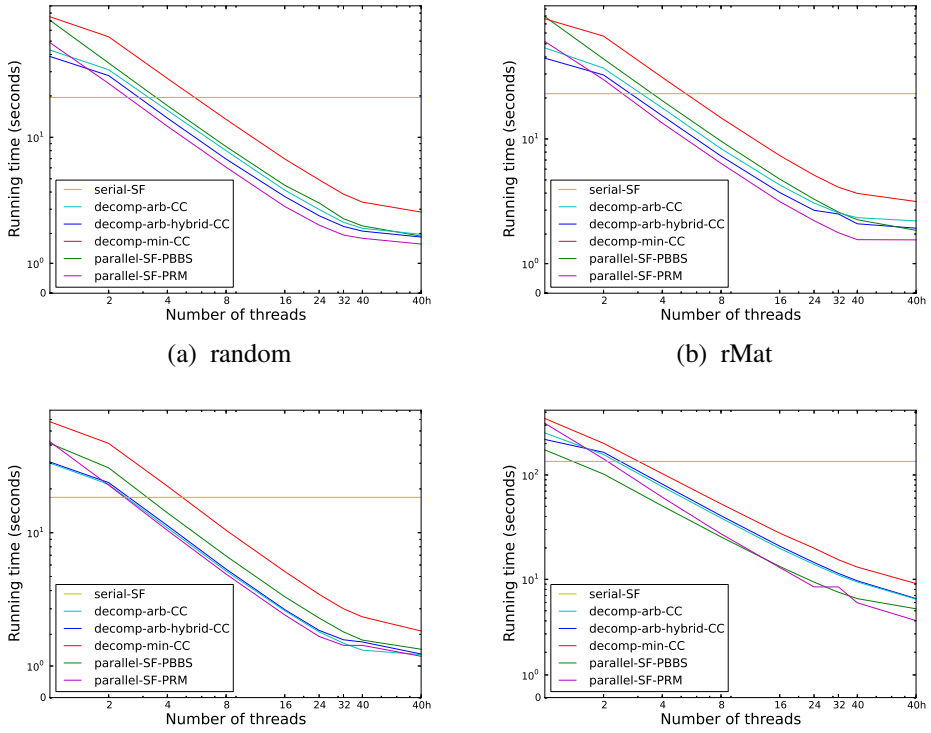
Table 5.2: Times (seconds) for connected components labeling. (40h) indicates 40 cores with hyper-threading. *We used the timing for the sequential spanning forest code from Patwary et al. [37] as we found it to be faster than the PBBS implementation.

Furthermore, decomp-arb-CC is usually outperformed by decomp-arb-hybrid-CC, due to the direction optimizing BFS that is implemented in decomp-arb-hybrid-CC. For graphs where the frontier grows large, the bottom-up BFS allows the algorithm to exploit more parallelism than the top-down BFS, which results in a performance increase for the hybrid algorithm. We note that the times for decomp-arb-CC and decomp-arb-hybrid-CC is roughly the same on the 3D-grid and line graphs, as the size of the frontier never grows large enough to warrant the read-based (bottom up) computation.

Finally, compared to parallel-SF-PRM – the lock based algorithm by Patwary et al. – our fastest algorithm, detomp-arb-hybrid-CC is at most 70% times slower in parallel, and is faster sequentially. We also note that compared to parallel-SF-PBBS, the second parallel algorithm, our code is faster in parallel on all inputs. Finally, compared to the serial implementation, decomp-arb-hybrid-CC is about 1.4–3.6 times slower on a single thread. The self-relative speedup of our fastest parallel implementation (decomp-arb-hybrid-CC) is 18–38. We achieve a speedup of 10–25 relative to the sequential implementation on this class of graphs.

We also plotted the running times displayed in Table 5.2 for a various thread counts in Figure 5.1. Both axes are plotted on log-scale. We also included the serial time for the sequential algorithm (serial-SF) in order to provide a baseline comparison. We observe that all parallel implementations achieve good speedup, and start to outperform the sequential algorithm even on a small number of threads. We also notice that the linear-work (work-efficient) implementations of parallel connectivity are not much slower than the spanning forest-based implementations, which are theoretically super-linear work algorithms.

We also investigated various properties of our decomposition algorithms as a function of β. Recall that $0 < \beta < 1$ is a parameter of the low-diameter decomposition algorithm which controls both the diameter of pieces in the decomposition, as well as the number of inter-component edges observed in the decomposition. We show a plot of β vs running-time in Figure 5.2 on 40-cores for decomp-arb-CC, decomp-arb-hybrid-CC, and decomp-min-CC. We noticed the similarity between the deterministic and non-deterministic algorithms, which is suggested by the theory presented in Section 4.2. Perhaps as expected, lower values of β correspond to more performant connectivity algorithms, which is likely due to the lower number of inter-piece edges



(a) random (b) rMat

(c) 3D-grid (d) line

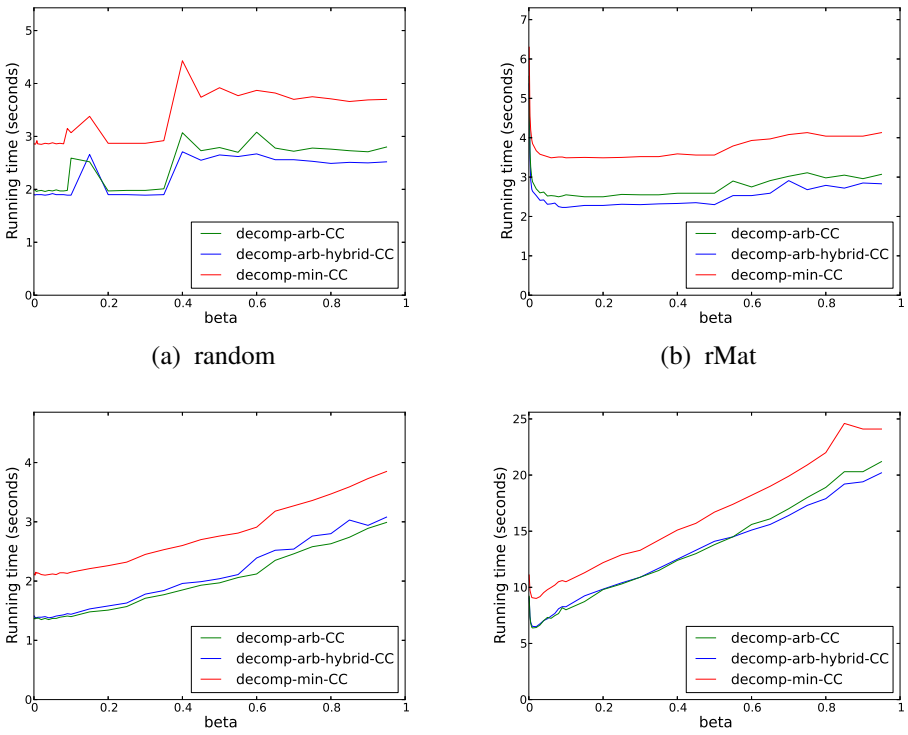
Figure 5.1: Times versus number of threads on various graphs on a 40-core machine with hyper-threading. (40h) indicates 80 hyper-threads.

passed down to recursive calls of the algorithm. We observed that a value of $\beta = 0.2$ provided good results across all graphs.

Finally, Figure 5.3 shows the number of edges remaining per iteration of the algorithm as a function of β. We plot this quantity for decomp-arb-hybrid-CC. As predicted by the theory, smaller values of β correspond to a fewer number of edges between pieces, which leads to fewer iterations until the algorithm terminates. We also note that the theoretical upper bound of $2\beta m$ on the number of inter-piece edges being removed is an upper-bound, and does not account for the large number of multi-edges that are eradicated by removing duplicate edges. In practice, this leads to a significantly sharper decrease in the number of edges removed than predicted by theory. It would be interesting to find a way to account for multi-edges in the theoretical bounds.

We also performed granular profiling of each of our three algorithms, decomp-arb-CC, decomp-arb-hybrid-CC, and decomp-min-CC.

In Figure 5.4, we show the breakdown for the 40-core times of decomp-min-CC (the deterministic algorithm). We break the running of the algorithm into 5 phases: init, bfsPre, bfsPhase1, bfsPhase2, and contractGraph. ***init*** accounts for the time used when generating random permutations and initializing arrays in all phases. ***bfsPre*** refers to the time used when adding vertices to the frontier, and computing offsets into shared arrays. ***bfsPhase1*** and ***bfsPhase2*** refer to the



(a) random (b) rMat

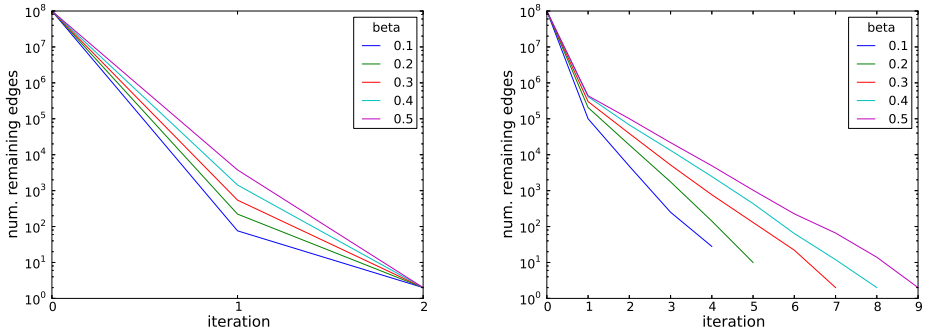
(c) 3D-grid (d) line

Figure 5.2: Running time versus β on various input graphs on a 40-core machine using 80 hyper-threads.

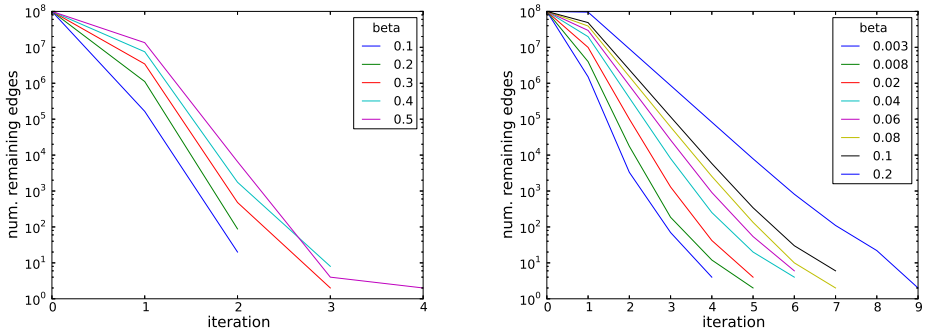
first BFS and second BFS respectively of Algorithm 2. Finally, ***contractGraph*** refers to the time spent removing duplicates, relabeling vertices, and creating the contracted graph.

Figure 5.5 displays the breakdown for the 40-c0re times for decomp-arb-CC. ***init***, ***bfsPre***, and ***contractGraph*** account for the same quantities as before. ***bfsMain*** is the time spent in the main phase of the BFS of Algorithm 3. We note that most of the running time of the algorithm is spent in the BFS. Compared to decomp-min-CC, the breakdown makes it clear that the primary cause for the improvement in running time is the elimination of the first BFS using the writeMin.

Figure 5.6 displays the breakdown for the 40-core times for decomp-arb-hybrid-CC. ***init***, and ***bfsPre*** account for the same quantities as before. ***bfsSparse*** is the time spent performing the top-down BFS using parallelism over the current frontier vertices. ***bfsDense*** is the time spent performing the bottom-up read based BFS by parallelizing over all unvisited vertices. ***filterEdges*** is the time spent filtering out intra-piece edges. We notice that for 3D-grid and the line graph, the frontier never becomes dense enough to use the dense read based BFS. We also note that because we still have to inspect all edges in order to compact out intra-piece edges, the graph on which the read-based BFS is actually used have to pay extra in the filterEdges phase in order to compact the adjacency lists.



(a) random (b) rMat



(c) 3D-grid (d) line

Figure 5.3: Number of remaining edges per iteration versus β of decomp-arb-hybrid-CC on various graphs.

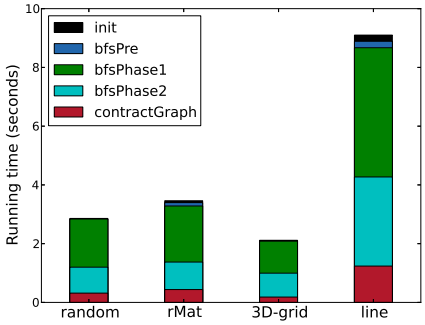


Figure 5.4: Breakdown of timings on 40 cores with hyper-threading for decomp-min-CC.

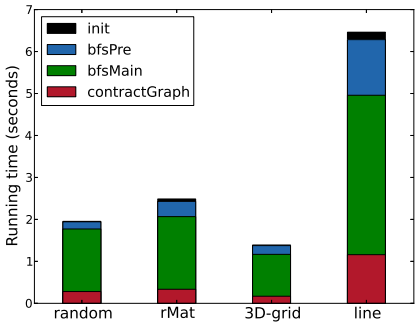


Figure 5.5: Breakdown of timings on 40 cores with hyper-threading for decomp-arb-CC.

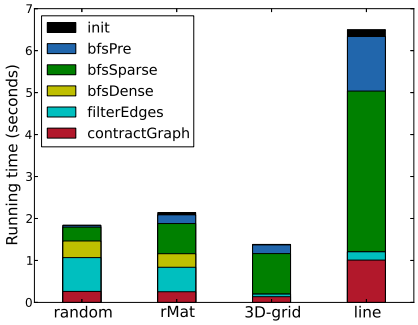


Figure 5.6: Breakdown of timings on 40 cores with hyper-threading for decomp-arb-hybrid-CC.

Chapter 6

Conclusion

The rapidly growing size of modern graphs has created a number of unique problems and opportunities for researchers. In order to be useful in this new modern landscape of massive graphs, theoretically work-efficient algorithms for the PRAM must be rethought and revisited. Despite their theoretical work-efficiency, many of these algorithms remain impractical to implement. Ongoing research at the intersection of the theory and practice of parallel algorithms must address these issues, while providing a solution that scales with the exponential increase in graph sizes observed yearly. To this end, we have made two contributions in this work, both of which address the problem of scaling algorithms for large graphs.

We have described and implemented *Cogra*, a compressed graph processing framework for shared memory architectures. Using Cogra, we can compress graphs to significantly fewer bits-per-edge, allowing researchers to fit previously prohibitively large graphs in main memory. The compressed representation illustrates the balance between lowering the number of bits-per-edge in the graph while still allowing reasonable query times on a given vertex's adjacency list. We also show how the compressed representation is empirically as fast as, or even faster than a framework operating on an uncompressed representation of the graph.

Approaching the scalability problem in a different direction, we have also described a new theoretically work-efficient and practical algorithm for computing the connected components and connectivity labeling of a graph. Unlike other existing implementations of parallel connectivity, our algorithm is grounded in theory. We also find an implementation of our algorithm to be competitive with the best known implementation of graph connectivity. The algorithm is the first known practical work-efficient parallel algorithm for this problem.

We conclude by remarking on the sheer number of problems that require attention from researchers interested in both the theory and practice of parallel algorithms. From graph-analytic problems such as inference and topic-recommendation to fundamental problems on graphs such as bi-connectivity and maximum-flow, there are an overwhelming amount of problems still waiting to be tackled from a highly parallel and scalable perspective. Future work must investigate these fundamental problems in order to design theoretically sound and simultaneously performant algorithms that are useful in practice.

Bibliography

[1] Stanford network analysis project (SNAP). Available at `http://snap.stanford.edu/`. 3.5, 5.3

[2] Baruch Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for Ultracomputer and PRAM. In *ICPP*, 1983. 4.1.1

[3] Scott Beamer, Krste Asanović, and David Patterson. Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500. *Technical Report UCB/EECS-2011-117, EECS Department, University of California, Berkeley*, 2011. 3.2.1, 4, 5.2

[4] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2012. 3.2.1

[5] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. Compact representations of separable graphs. In *SODA*, pages 679–688, 2003. 1, 2, 2, 2.2.1, 2.2.2

[6] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. An experimental analysis of a compact graph representation. In *ALENEX*, pages 49–61, 2004. 2, 2.3

[7] Guy E. Blelloch, Anupam Gupta, Ioannis Koutis, Gary L. Miller, Richard Peng, and Kanat Tangwongsan. Near linear-work parallel SDD solvers, low-diameter decomposition, and low-stretch subgraphs. In *SPAA*, 2011. 4.2

[8] Paolo Boldi and Sebastiano Vigna. The webgraph framework I: compression techniques. In *WWW*, pages 595–602, 2004. 2, 1, 2

[9] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A large time-aware graph. *SIGIR*, 42 (2):33–38, 2008. 3.5

[10] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web*, pages 587–596, 2011. 2.2

[11] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25, 2001. 3.4

[12] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, 1998. 3.4

[13] Aydın Buluç and John R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*,

2011. 3

[14] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, 2004. 1.1

[15] C. Chevalier and F. Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(68):318 – 331, 2008. 2.2.2

[16] Richard Cole, Philip N. Klein, and Robert Endre Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *SPAA*, 1996. 4.1.3

[17] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, November 2011. 3.5

[18] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, Mar 1975. 2.3

[19] Hillel Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, 20(6), December 1991. 4.1.3

[20] Giraph, 2012. http://giraph.apache.org. 1

[21] Jean-Loup Guillaume, Matthieu Latapy, et al. The web graph: an overview. In *Actes d'ALGOTEL'02 (Quatrièmes Rencontres Francophones sur les aspects Algorithmiques des Télécommunications)*, 2002. 1, 3.2.2

[22] Shay Halperin and Uri Zwick. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM. *J. Comput. Syst. Sci.*, 53(3), 1996. 4.1.3

[23] Shay Halperin and Uri Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests. In *J. Algorithms*, 2000. 4.1.3

[24] Guy Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554, 1989. 1

[25] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2), 2011. 3.1

[26] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 1998. 2.2.2

[27] Charles E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3), 2010. 1.1

[28] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence*, 2010. 3.1, 4.1.4

[29] Adam Lugowski, David Alber, Aydın Buluç, John Gilbert, Steve Reinhardt, Yun Teng, and Andrew Waranis. A flexible open-source toolbox for scalable complex graph analysis. In *SDM*, 2012. 3, 3.1

[30] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010. 1, 3.1

[31] Yossi Matias and Uzi Vishkin. On parallel hashing and integer sorting. *Journal of Algorithms*, 12(4), 1991. 5.1.1

[32] Gary L. Miller, Richard Peng, and Shen Chen Xu. Parallel graph decomposition using random shifts. In *SPAA*, 2013. 4.2, 4.2, 4.2, 4.3, 5, 5.1, 5.2

[33] Moni Naor. Succinct representation of general unlabeled graphs. *Discrete Applied Mathematics*, 28(3):303 – 307, 1990. 2

[34] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Grappa: A latency-tolerant runtime for large-scale irregular applications. Technical report, Technical Report UW-CSE-14-02-01, University of Washington, 2014. 3.1

[35] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007. 3.5.2

[36] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471, 2013. 3.1

[37] M.M.A. Patwary, P. Refsnes, and F. Manne. Multi-core spanning forest algorithms using the disjoint-set data structure. In *IPDPS*, 2012. 4.1.1, 5.3, 5.2

[38] Francois Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 493–498. 1996. 2.2.2

[39] Seth Pettie and Vijaya Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J. Comput.*, 31(6), 2002. 4.1.3

[40] C. A. Phillips. Parallel graph contraction. In *SPAA*, 1989. 4.1.2

[41] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *PLDI*, 2011. 3.1

[42] Chung Keung Poon and Vijaya Ramachandran. A randomized linear work EREW PRAM algorithm to find a minimum spanning forest. In *ISAAC*, 1997. 4.1.3

[43] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. Managing large graphs on multi-cores with graph awareness. In *USENIX ATC*, 2012. 3.1

[44] Sriram Raghavan and Hector Garcia-Molina. Representing web graphs. In *ICDE*, pages 405–416, 2003. 2

[45] Keith H. Randall, Raymie Stata, Janet L. Wiener, and Rajiv G. Wickremesinghe. The link database: Fast access to graphs of the web. In *Proceedings of the Data Compression Conference*, 2002. 2

[46] J. Reif. Optimal parallel algorithms for integer sorting and graph connectivity. *TR-08-85, Harvard University*, 1985. 4.1.2

[47] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: edge-centric graph

processing using streaming partitions. In *24th ACM Symposium on Operating Systems Principles*, pages 472–488, New York, 2013. ACM. 3.1

[48] Semih Salihoglu and Jennifer Widom. GPS: A graph processing system. Technical Report InfoLab 1039, Stanford University, 2012. 1, 3.1

[49] O. Schenk, A. Wchter, and M. Weiser. Inertia-revealing preconditioning for large-scale nonconvex constrained optimization. *SIAM Journal on Scientific Computing*, 31(2):939–960, 2009. 3.5

[50] Yossi Shiloach and Uzi Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3(1), 1982. 4.1.1

[51] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared-memory. In *PPoPP*, 2013. 4, 3.4, 4.1.4

[52] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *SPAA*, 2012. 5.3

[53] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *SPAA*, 2012. 1.1, 2.2.2

[54] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Reducing contention through priority updates. In *SPAA*, 2013. 3.4

[55] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3):193–204, 2013. 2

[56] G. Turan. Succinct representation of graphs. *Discrete Applied Mathematics*, 8:289–294, 1984. 2

[57] Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013. 3.1

[58] Wenlei Xie, Guozhang Wang, David Bindel, Alan J. Demers, and Johannes Gehrke. Fast iterative graph computation with block updates. *PVLDB*, 6(14), 2013. 3.1

[59] Yahoo! Altavista web page hyperlink connectivity graph, 2012. "http://webscope.sandbox.yahoo.com/catalog.php?datatype=g". 1, 3.5

[60] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978. 2