# GraphCT: Multithreaded Algorithms for Massive Graph Analysis

David Ediger, *Member, IEEE*, Karl Jiang, E. Jason Riedy, *Member, IEEE*, and David A. Bader, *Fellow, IEEE*

**Abstract**—The digital world has given rise to massive quantities of data that include rich semantic and complex networks. A social graph, for example, containing hundreds of millions of actors and tens of billions of relationships is not uncommon. Analyzing these large datasets, even to answer simple analytic queries, often pushes the limits of algorithms and machine architectures. We present GraphCT, a scalable framework for graph analysis using parallel and multithreaded algorithms on shared memory platforms. Utilizing the unique characteristics of the Cray XMT, GraphCT enables fast network analysis at unprecedented scales on a variety of input datasets. On a synthetic power law graph with 135 million vertices and 4 billion edges, we can find the connected components in under 3 minutes. We can estimate the betweenness centrality of a similar graph with 537 million vertices and over 8 billion edges in under one hour. GraphCT also builds on multicore systems using OpenMP.

**Index Terms**—Graph algorithms, network analysis, Cray XMT, multithreaded architectures, high-performance computing.

## 1 Introduction

The vast quantity of data being created by social networks [1], sensor networks [2], healthcare records [3], bioinformatics [4], computer network security [5], computational sciences [6], and many other fields offers new challenges for analysis. When represented as a graph, this data can fuel knowledge discovery by revealing significant interactions and community structures. Current network analysis software packages (e.g. Pajek [7], R (igraph) [8], Tulip [9], UCInet [10]) can handle graphs up to several thousand vertices and a million edges. These applications are limited by the scalability of the supported algorithms and the resources of the workstation. In order to analyze today's graphs and the semantic data of the future, scalable algorithms and machine architectures are needed for data-intensive computing. GraphCT is a collection of new parallel and scalable algorithms for static graph analysis. These algorithms, running atop multithreaded architectures such as the Cray XMT and Intel multicore CPUs, can analyze graphs with hundreds of millions of vertices and billions of edges in minutes, instead of days or weeks. GraphCT is able to, for the first time, enable analysts and domain experts to conduct in-depth analytical workflows of their data at massive scales in an interative time frame.

The foundation of GraphCT is a modular kernel-based design using efficient data representations in which an analysis workflow can be expressed through a series of function calls. All functions are required to use a single graph data representation. While some algorithms or input graphs may benefit from a particular data representation, the use of a single common data structure enables plug-and-play capability as well as ease of implementation and sharing new kernels. Basic data input/output as well as fundamental graph operations such as subgraph extraction are provided to enable domain scientists to focus on conducting high-level analyses. A wide variety of multithreaded graph algorithms are provided including clustering coefficients, connected components, betweenness centrality, $k$-core, and others, from which workflows can easily be developed. Figure 2 illustrates an example workflow. Analysis can be conducted on unweighted and weighted graphs, undirected and directed. Using the included compatibility layer, GraphCT can be built on the Cray XMT with XMT-C or on a commodity Intel or AMD workstation

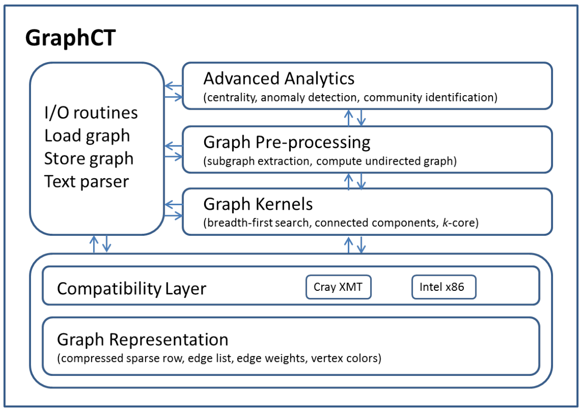

GraphCT
I/O routines
Load graph
Store graph
Text parser
Advanced Analytics
(centrality, anomaly detection, community identification)
Graph Pre-processing
(subgraph extraction, compute undirected graph)
Graph Kernels
(breadth-first search, connected components, k-core)
Compatibility Layer
Cray XMT
Intel x86
Graph Representation
(compressed sparse row, edge list, edge weights, vertex colors)

Fig. 1. GraphCT is an open source framework for developing scalable multithreaded graph analytics in a cross-platform environment.

- *The authors are with the School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA, 30332.*

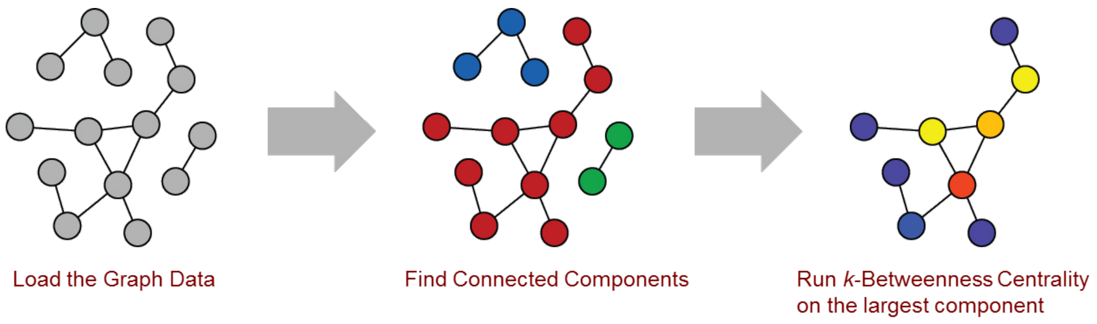

Load the Graph Data
Find Connected Components
Run k-Betweenness Centrality on the largest component

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

using OpenMP.

The near-exponential growth of massive social networks on the Internet over the last several years has been staggering. Facebook has more than 845 million users, over half of which are active daily [1]. Twitter has tens of millions of users, and the blogosphere has an estimated hundreds of millions of English language blogs. In each case, the network contains both topological information (actors and links) as well as a rich semantic network of interactions. If the topology information of Facebook alone was represented with a compressed sparse row plus edge list and edge weights using 64-bit data types, the data structure alone would cost over 2.4 TB of memory. If a machine could iterate over 1 billion edges per second, it would take 2 minutes to read each edge one time. The scale of these social networks necessitates specialized computer architecture and massively parallel algorithms for analysis.

Real world networks of this kind challenge modern computing in several ways. These graphs typically exhibit “small-world” [11] properties such as small diameter and skewed degree distribution. The low diameter implies that all reachable vertices can be found in a small number of hops. A highly skewed degree distribution, where most vertices have a few neighbors and several vertices have many neighbors, often leads to workload imbalance among threads. One proposed solution is to handle high- and low-degree vertices separately; parallelize work across low-degree vertices and within high-degree vertices. A runtime system must be able to handle dynamic, fine-grained parallelism among hundreds of thousands of threads with low overhead. Executing a breadth-first search from a particular vertex quickly consumes the entire graph. The memory access pattern of such an operation is unpredictable with little spatial or temporal locality. Caches are ineffective for lowering memory access latency in this case. The Cray XMT relies on hardware multithreading with low overhead context switches, rather than caches, to tolerate the latency to memory. Lightweight and fine-grained synchronization enable algorithm designers to expose large amounts of parallelism in the application.

In our previous work [12], [13], [14], [15] we presented new parallel algorithms for the analysis of online social networks and implementations on small multithreaded architectures, such as Intel Nehalem, Sun Niagara, and small Cray XMTs. We have extended our work to scale up to larger machines (up to 128 processors) using a wider range of graph types including larger graph datasets and graphs originating from real-world data sources. We have added a number of new multithreaded algorithms for massive graph analysis in a simple, yet powerful framework.

The main contributions of this work are:

- $k$-Betweenness Centrality, a new parallel algorithm for finding important vertices in a network that is robust to edge deletions
- Unprecedented scalability of complex analytics to 128 processors
- The first analysis framework to enable a workflow of analytics on graphs with billions of vertices and edges

In the remainder of the paper, we will present the design challenges and experimental results for GraphCT. Section 2 will cover the requirements for this graph application. Section 3 will explain the various kernels and key features that have been developed. Sections 4, 5, and 6 will present the implementation and performance of connected components, clustering coefficients, and $k$-betweenness centrality, respectively.

## 2 Motivation and Requirements

A number of software applications have been developed on the desktop computer for analyzing and visualizing graph datasets. Among them, Pajek is one of the most widely used, along with R (igraph), Tulip, UCInet, and many others [7], [8], [9], [10]. While each application has its differences, all are limited by the size of workstation main memory. Pajek has been known to run complex graph analytics on inputs with up to two million vertices, but many other applications are limited to tens or hundreds of thousands of vertices.

In the 9th DIMACS challenge on shortest path routing, a road network consisting of the entire United States of America had almost 24 million vertices and 60 million edges [16]. The Facebook social network "friend" graph is estimated to contain approximately 845 million vertices and average degree of 130 [1]. From these examples, it should be clear that real data analysis requires large parallel machines and scalable software.

The development of new scalable algorithms and frameworks for massive graph analysis is the subject of many research efforts. The Parallel Boost Graph Library (PBGL) [17] is a C++ library for distributed memory graph computations. The API is similar to that of the Boost Graph Library. The authors report scalable performance up to about 100 processors. Distributed memory graph processing often requires partitioning and data replication, which can be challenging for some classes of graphs.

SNAP [18] is an open source parallel library for network analysis and partitioning using multicore workstations. It is parallelized using OpenMP and provides a simple API with support for very large scale graphs. It is one of the only libraries that provides a suite of algorithms for community detection.

Sandia's Multithreaded Graph Library (MTGL) [19] is a C++ library for implementing graph applications on multithreaded architectures, particularly the Cray XMT. MTGL uses the notion of a "visitor" class to describe operations that take place when a vertex is visited, such as during a breadth first search.

Given the immense size in memory of the graphs of interest, it is not possible to store a separate representation for each analysis kernel. Since the key capability of GraphCT is running a number of analytics against an unknown data source, we employ a simple, yet powerful framework for our computation. Each kernel implementation is required to use a common graph data structure. Using an application-specific representation may accelerate certain algorithms, but also induces a cost in terms of computation and memory footprint to translate between representations as one moves from kernel to kernel. By using the same data structure for each kernel, all kernels can be run in succession (or even in parallel if resources allow) without the need to transfer the graph between data structures. As a result, GraphCT enables the results of one computation to easily influence the next computation, such as the extraction of one or more connected components for more in-depth study.

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

## 2.1 The Cray XMT

To facilitate scaling to the sizes of massive datasets previously described, GraphCT will utilize the massive shared memory and multithreading capabilities of the Cray XMT. Large planar graphs, such as road networks, can be partitioned with small separators and analyzed in distributed memory with good computation-to-communication ratios at the boundaries. Graphs arising from massive social networks, on the other hand, are challenging to partition and lack small separators [18], [20]. For these problems, utilizing a large global shared memory eliminates the requirement that data must be evenly partitioned. The entire graph can be stored in main memory and accessed by all threads. With this architectural feature, parallelism can be expressed at the level of individual vertices and edges. Enabling parallelism at this level requires fine-grained synchronization constructs such as atomic fetch-and-add and compare-and-swap.

The Cray XMT offers a global shared memory using physically distributed memories interconnected by a high speed, low latency, proprietary network. Memory addresses are hashed to intentionally break up locality, effectively spreading data throughout the machine. As a result, nearly every memory reference is a read or write to a remote memory. Graph analysis codes are generally a series of memory references with very little computation in between, resulting in an application that runs at the speed of memory and the network.

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

Since a 128-processor Cray XMT contains about 12,000 user hardware contexts, it is the responsibility of the programmer to reveal a large degree of parallelism in the code. Coarse- as well as fine-grained parallelism can be exploited using Cray's parallelizing compiler. The programmer inserts `#pragma` statements to assert that a given loop's iterations are independent. Often iterations of a loop will synchronize on shared data. To exploit this kind of parallelism, the Cray XMT provides low-cost fine-grained synchronization primitives such as full-empty bit synchronization and atomic fetch-and-add. Using these constructs, it is possible to expose fine-grained parallelism, such as operations over all vertices and all neighbors, as well as coarse-grained parallelism, such as multiple breadth first searches in parallel. Internally, the fine-grained synchronization constructs are used by the Cray runtime for dynamic load balancing, which helps to alleviate problems associated with the highly skewed degree distribution of scale-free networks.

## 3 Features

### 3.1 Data Representation

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

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

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

In this format, we can represent both directed and undirected graphs. The common data representation between kernels relieves some of the burden of allocating frequently used in-memory data structures. With the graph remaining in-memory between kernel calls, we provide a straightforward API through which analytics can communicate their results.

### 3.2 Clustering Coefficients

Clustering coefficients measure the density of closed triangles in a network and are one method for determining if a graph is a small-world graph [11]. We may be interested in the global clustering coefficient, which is a single number describing the entire graph, or the local clustering coefficients, which is a per-vertex measure of triangles. Although there is some disagreement in the literature, both of these metrics are applicable to undirected graphs. For directed graphs, several variations have been proposed and we have adopted the transitivity coefficients, which is a natural extension of the local clustering coefficient idea.

Section 5 contains a detailed case study of our implementation on the Cray XMT and performance results on large synthetic networks.

### 3.3 Connected Components

The connected components of the graph indicate the existence of paths between sets of vertices. If two vertices are in the same component, then there exists a path between them. Likewise, if two vertices reside in different components, a search from one vertex will not find the other. If the connected components of the graph are known, determining the $st$ connectivity for a pair of vertices can be calculated easily.

In Section 4 we will offer in-depth coverage of the algorithm, implementation, and performance of our connected components routine. In short, we use a shared memory version of the classical Shiloach and Vishkin algorithm. On the Cray XMT, we determine the connected components of a scale-free undirected graph with 135 million vertices and 2 billion edges in about 15 seconds.

### 3.4 Distributions

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

The maximum degree, average degree, and variance are quickly calculated using a single loop and several accumulators over the vertex offset array. On the

Cray XMT, the compiler is able to easily parallelize this loop. The same technique can be applied to the output of the connected components function. Given the vertex-component mapping, we must first histogram the values to determine the number of vertices in each component, but then can, in the same manner, calculate distribution statistics on the size of connected components.

### 3.5 Graph Diameter

The diameter of the graph is an important metric for understanding the nature of the input graph at first glance. If interested in the spread of disease in an interaction network, the diameter may helpful in estimating the rate of transmission and the time to full coverage. Calculating the diameter exactly requires an all-pairs shortest path computation, which is prohibitive for the large graphs of interest. Estimating the graph diameter is a well-studied problem and many approaches have been suggested.

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

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

### 3.6 Graph Parsing and Generation

Since the purpose of graph analysis in general, and GraphCT in particular, is to shed light on the characteristics of real datasets, we must provide mechanisms to import data from the outside world. Given that the graphs are so large as to require machines with terabytes of main memory, it would be reasonable to expect the input data files to be of the same massive size. Most existing graph libraries for the desktop workstation import file formats that are essentially edge lists in text format. To support a similar mechanism, we must be able to read from disk an entire text file, read each line (corresponding to an edge), and construct a graph data structure in memory.

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

In the event that an input dataset is likely to be used frequently, we provide an offline mechanism to parse the text file and output an efficient binary representation of the internal data structure. This binary file on disk is likely to be much smaller than the original text file, reducing storage requirements and I/O time.

For instances when real data is not available with the scale or characteristics of interest, GraphCT is able to provide graph generators that will provide an output file on disk that can be read in for kernel testing. As an example, we provide an implementation of the RMAT graph generator, which was used in the DARPA High Productivity Computing Systems (HPCS) Scalable Synthetic Compact Applications benchmark #2 (SSCA2). This generator uses repeated sampling from a Kronecker product to produce a random graph with a degree distribution similar to those arising from social networks. By varying in input parameters of the generator, it can also be used to produce an Erdös-Rényi random graph. The generator takes as input the probabilities $a$, $b$, $c$, and $d$ which determine the degree distribution, the number of vertices (must be a power of two), and the number of edges. Duplicate and self edges are removed, so the total number of edges produced may be slightly less than the input parameter given.

### 3.7 Modularity and Conductance

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

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

where $i$ and $j$ are vertices in the graph, $m$ is the total number of edges, $k_i$ is the degree of vertex $i$, and $s_i$ expresses the community to which vertex $i$ belongs [21].

Given a community mapping of vertices, modularity is calculated using two parallel loops over all vertices. The first calculates the total number of edges in each community. The second loop gives credit for neighbors of vertices that are in the same community and subtracts credit for the external connections. The modularity score is reported and returned at the end. This function can be easily used as a scoring component of a clustering method. For example, in greedy agglomerative clustering, after each component merge the modularity is evaluated and stored in the merge tree.

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

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

where $\bar{S}$ is the set of vertices not in $S$ and $e(S, \bar{S})$ is the number of edges between $S$ and $\bar{S}$. The total number of edges that could span the cut is expressed as $d\,|S| = \sum_{v \in S} d(v)$ where $d(v)$ is the degree of vertex $v$ [22]. The value of $\Phi$ ranges from 0 to 1. While the formula above is for an unweighted graph, it can be generalized to weighted networks by summing edge weights instead of degrees.

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

## 4 CONNECTED COMPONENTS

Finding the connected components of the graph determines a per-vertex mapping such that all vertices in a component are reachable from each other and not reachable from those vertices in other components. Understanding the size distribution of components as well as the number of connected components is useful for first order analysis of a graph. The connected components mapping is also useful for other analytics. A sampling algorithm may sample vertices according to the distribution of component sizes such that all components are appropriately represented in the sampling. An analysis may focus on just the small components or only the biggest component in order to isolate those vertices of greatest interest. Finding the connected components is a useful metric unto itself and also a common pre-processing step.

Fig. 3. Parallel multithreaded version of Shiloach-Vishkin algorithm for finding the connected components of a graph.

```
Input: $G(V, E)$
Output: $M[1..n]$, where $M[v]$ is the component to
    which vertex $v$ belongs
1: $changed \leftarrow 1$
2: for all $v \in V$ in parallel do
3:   $M[v] \leftarrow v$
4: while $changed \neq 0$ do
5:   $changed := 0$
6:   for all $\langle i, j \rangle \in E$ in parallel do
7:     if $M[i] < M[j]$ then
8:       $M[M[j]] := M[i]$
9:       $changed := 1$
10:  for all $v \in V$ in parallel do
11:    while $M[v] \neq M[M[v]]$ do
12:      $M[v] := M[M[v]]$
```

The Shiloach and Vishkin algorithm [23] is a classical algorithm for finding the connected components of an undirected graph. This algorithm is well suited for shared memory and exhibits per-edge parallelism that can be exploited.

Each vertex is initialized to its own unique color. At each step, neighboring vertices greedily color each other such that the vertex with the lowest ID wins. The process ends when each vertex is the same color as its neighbors. The number of steps is proportional to the diameter of the graph, so for small-world networks the algorithm converges quickly.

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

In Figure 4 we present execution times for connected components on several massive undirected graphs using a 128 processor Cray XMT. The first graph is a sparse, planar graph of the US road network. The rest of the graphs are synthetic RMAT graphs with small-world properties. Using 128 pro-

| Name | NV | NE | P=16 | P=32 | P=64 | P=128 |
|---|---|---|---|---|---|---|
| USA Road Network | 23,947,347 | 58,333,344 | 4.13 | 3.17 | 2.64 | 4.26 |
| RMAT | 134,217,727 | 1,071,420,576 | 34.7 | 21.0 | 13.8 | 12.3 |
| RMAT | 134,217,727 | 2,139,802,777 | 59.3 | 36.8 | 20.0 | 15.1 |
| RMAT | 134,217,727 | 4,270,508,334 | 251.6 | 211.7 | 110.2 | 164.3 |

Fig. 4. Running times in seconds for connected components on a 128 processor Cray XMT

cessors, we can quickly determine the connected components of the graph in under 20 seconds. Scalability increases as the density of the network increases.

## 5 Clustering Coefficients

Clustering coefficients measure the density of closed triangles in a network and are one method for determining if a graph is a small-world graph [11]. We adopt the terminology of [11] and limit our focus to *undirected* and unweighted graphs. A triplet is an ordered set of three vertices, $(i, v, j)$, where $v$ is considered the focal point and there are undirected edges $\langle i, v\rangle$ and $\langle v, j\rangle$. An open triplet is defined as three vertices in which only the required two are connected, for example the triplet $(m, v, n)$ in Figure 5. A closed triplet is defined as three vertices in which there are three edges, or Figure 5's triplet $(i, v, j)$. A triangle is made up of three closed triplets, one for each vertex of the triangle.

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

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

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

$$C_v = \frac{\text{number of closed triplets around } v}{\text{number of triplets around } v}. \quad (4)$$

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

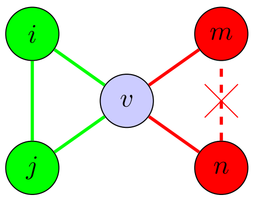


Fig. 5. There are two triplets around $v$ in this unweighted, undirected graph. The triplet $(m, v, n)$ is open, there is no edge $\langle m, n\rangle$. The triplet $(i, v, j)$ is closed.

$$C_v = \frac{\sum_{i \in e_v} |e_i \cap (e_v \setminus \{v\})|}{d_v(d_v - 1)} = \frac{T_v}{d_v(d_v - 1)}. \quad (5)$$

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

Our test data is generated by sampling from a Kronecker product using the RMAT recursive matrix generator [24] with probabilities $A = 0.55$, $B = 0.1$, $C = 0.1$, and $D = 0.25$. Each generated graph has a few vertices of high degree and many vertices of low degree. Given the RMAT scale $k$, the number of vertices $n = 2^k$, and an edge factor $f$, we generate approximately $f \cdot n$ unique edges for our graph.

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

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

In the plot on the right, the same kernel is run on the USA road network, a graph with 24 million vertices and 58 million edges. The graph is nearly planar with a small, uniform degree distribution. Because the amount of work per vertex is nearly equal, the vertices are easily scheduled and the algorithm scales linearly to 128 processors. The total execution time is about 27 seconds.

These results highlight the challenges of developing scalable algorithms on massive graphs. Where commodity platforms often struggle to obtain a speed-up, the latency tolerance and massive multithreading of the Cray XMT enables linear scalability on regular, uniform graphs. The discrepancy between the

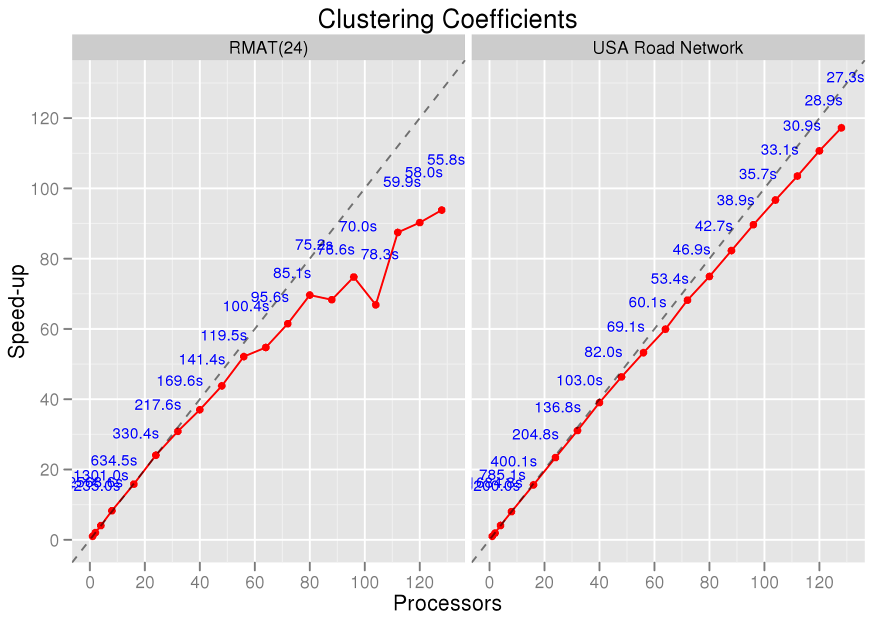


Fig. 6. Scalability of the local clustering coefficients kernel on the Cray XMT. On the left, the input graph is an undirected RMAT generated graph with approximately 16 million vertices and 135 million edges. The speedup is 94x on 128 processors. On the right, the input graph is the USA road network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

RMAT scalability (left) and the road network (right) is an artifact of the power law degree distribution of the former. Despite the complex and irregular graph topology, GraphCT is still able to scale up to 128 processors.

There are several variations of clustering coefficients for directed graphs. A straightforward approach is to apply the definition directly and count the number of triangles, where a triangle now requires six edges instead of three. A more sophisticated approach is called the transitivity coefficients. Transitivity coefficients counts the number of transitive triplets in the numerator. A transitive triplet is one in which edges exist from vertex $a$ to vertex $b$ and from vertex $b$ to vertex $c$, with a shortcut edge from vertex $a$ to vertex $c$ [25].

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

## 6 $k$-BETWEENESS CENTRALITY

Betweenness centrality has proved a useful analytic for ranking important vertices and edges in large graphs. As defined by Freeman in [26], betweenness centrality is a measure of the number of shortest paths

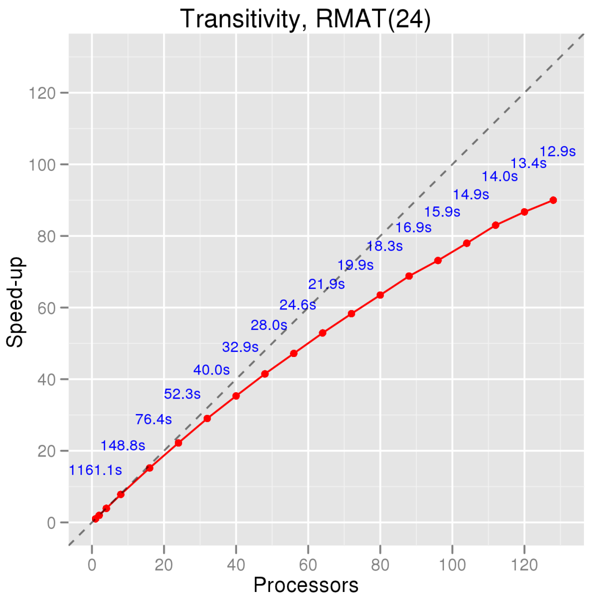


Fig. 7. Scalability of the transitivity coefficients kernel on the Cray XMT. The input graph is a directed RMAT generated graph with approximately 16 million vertices and 135 million edges. Execution times in seconds are shown in blue. On 128 processors, we achieve a speed-up of 90x.

in a graph passing through a given vertex. For a graph $G(V, E)$, let $\sigma_{st}$ denote the number of shortest paths between vertices $s$ and $t$, and $\sigma_{st}(v)$ the count of shortest paths that pass through a specified vertex $v$. The betweenness centrality of $v$ is defined as:

$$BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{6}$$

Betweenness centrality can be used to identify critical vertices in a network. High centrality scores indicate that a vertex lies on a considerable fraction of shortest paths connecting pairs of vertices. This metric has been applied extensively to the study of various networks including biological networks [4], sexual networks and the transmission of the AIDS virus [3], identifying key actors in terrorist networks [27], organizational behavior, and transportation networks [28].

In earlier work, we contributed a lock-free parallel algorithm for betweenness centrality [12]. In the remainder of this section, we will motivate and present an extension of Freeman's betweenness centrality and our previous algorithm. We generalize the definition to include paths in the graph whose length is within a specified value $k$ of the length of the shortest path. We extend our recent parallel, lock-free algorithm for computing betweenness centrality to compute generalized $k$-betweenness centrality scores. Next, we will give details of an implementation of our new algorithm on the massively multithreaded Cray XMT and describe the performance effects of this extension in terms of execution time and memory usage. Last, we will compare the results of this algorithm on synthetic and real-world datasets.

## 6.1 Extending Betweenness Centrality

The traditional definition of betweenness centrality [26] enumerates all shortest paths in a graph and defines betweenness centrality in terms of the ratio of shortest paths passing through a vertex $v$. This metric has proven valuable for a number of graph analysis applications, but fails to capture the robustness of a graph. A vertex that lies on a number of paths whose length is just one greater than the shortest path receives no additional value compared to a vertex with an equally large number of shortest paths, but few paths of length one greater.

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

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

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

## 6.2 A Parallel Algorithm for $k$-Betweenness Centrality

Brandes offered the first algorithm for computing betweenness in O$(mn)$ time for an unweighted graph [29]. Madduri and Bader developed a parallel betweenness algorithm motivated by Brandes' approach that exploits both coarse- and fine-grained parallelism in low-diameter graphs in [30] and improved the performance of this algorithm using lock-free methods in [12]. Here we extend the latter work to incorporate our new analytic of $k$-betweenness centrality.

We define $d(s, t)$ to be the length of the shortest path between $s$ and $t$. Paths must be acyclic and directed outwards from the source vertex. Let $\tau_{st_k}$ be the number of paths between $s$ and $t$ with length equal to exactly $d(s, t) + k$, and let $\tau_{st_k}(v)$ be the number of these paths that pass through vertex $v$. Then, $\tau_{st_k}(v)$ is given by:

$$\tau_{st_k}(v) = \sum_{i=0}^{k} \tau_{sv_i} \cdot \tau_{vt_{k-i}} \tag{8}$$

Clearly, for $k = 0$ (where $\tau_{st_0} = \sigma_{st_0}$ by definition), we have reproduced the original value of $\sigma_{st}(v)$ from Brandes where $d(s, t) \geq d(s, v) + d(v, t)$ (the Bellman criterion).

The $k$-betweenness centrality of vertex $v$ may be obtained by summing the pair-dependencies for that vertex:

$$BC_k(v) = \sum_{i=0}^{k} \sum_{s \neq v \neq t \in V} \delta_{st_k}(v) \tag{9}$$

$\delta_{st_k}(v)$ is given by the ratio of the number of paths whose length is equal to $d(s, t) + k$ passing through vertex $v$ over the total count of the paths of length less than or equal to $d(s, t) + k$ between $s$ and $t$:

$$\delta_{st_k}(v) = \frac{\tau_{st_k}(v)}{\sigma_{st_k}} \tag{10}$$

In his work, Brandes derives a recursive relation for the *dependence* of $s$ on any other vertex $v$ in the graph. Likewise, we have derived the general expression for any path length $k$ greater than the shortest path. We define $\Delta D(w, v)$ as $d(s, v) - d(s, w) + 1$, where $s$ is the source vertex and $w$ is a neighbor of $v$. $\Delta D$ is bounded by $k$ for neighbors lying on a $+k-$path in which we are interested. We define $\delta_{s_k}(v)$, the dependenc of $s$ on $v$ through paths of length $d(s, t) + k$, to be:

$$\delta_{s_k}(v) = \sum_{t \in V, t \neq s} \delta_{st_k}(v) \tag{11}$$

Fig. 8. A level-synchronous parallel algorithm for computing $k$-betweenness centrality of vertices in unweighted graphs.

```
Input: $G(V, E), k$
Output: $kBC[1..n]$, where $kBC[v]$ gives the $k$-centrality score ($BC_k(v)$) for vertex $v$
1: for all $v \in V$ in parallel do
2:   $kBC[v] \leftarrow 0$
3: for all $s \in V$ do
     I. Initialization
4:   for all $t \in V$ in parallel do $d[t] \leftarrow -1$
5:     for $0 \leq i \leq k$ in parallel do
6:       $Succ[i][t] \leftarrow$ empty multiset, $\tau[i][t] \leftarrow 0$,
7:   $\tau[0][s] \leftarrow 1$, $d[s] \leftarrow 0$
8:   $phase \leftarrow 0$, $S[phase] \leftarrow$ empty stack
9:   push $s \rightarrow S[phase]$
```

Fig. 9. Part II - Graph traversal for shortest path discovery and counting

```
1: $count \leftarrow 1$
2: while $count > 0$ do
3:   $count \leftarrow 0$
4:   for all $v \in S[phase]$ in parallel do
5:     for each neighbor $w$ of $v$ in parallel do
6:       if $d[w] < 0$ then
7:         push $w \rightarrow S[phase + 1]$
8:         $count \leftarrow count + 1$
9:         $d[w] \leftarrow d[v] + 1$
10:      $\Delta D = d[v] - d[w] + 1$
11:      if $\Delta D \leq min(k, 1)$ then
12:        $\tau[\Delta D][w] \leftarrow \tau[\Delta D][w] + \tau[0][v]$
13:      if $\Delta D \leq k$ then
14:        append $w \rightarrow Succ[\Delta D][v]$
15:  $phase \leftarrow phase + 1$
16: for $1 \leq i \leq k$ do
17:   for $0 \leq p < phase$ do
18:     for all $v \in S[p]$ in parallel do
19:       for all $w \in Succ[0][v]$ in parallel do
20:         $\tau[i][w] \leftarrow \tau[i][w] + \tau[i][v]$
21:       if $i < k$ then
22:         for $0 < j \leq i + 1$ in parallel do
23:           for all $w \in Succ[j][v]$ in parallel do
24:             $\tau[i+1][w] = \tau[i+1][w] + \tau[i+1-j][v]$
```

$$BC_k(v) = \sum_{i=0}^{k} \sum_{s \in V} \delta_{s_k}(v) \tag{12}$$

It follows that $BC_k(v)$ can be directly calculated from a sum of these dependence values. For the complete details of the derivation of $k$-betweenness centrality, please refer to the original paper [13].

An algorithm for $k$-betweenness centrality is given in Figures 8, 9, and 10. In the first stage we calculate $\tau_{sv_k}$, $\forall\ v, k$ from a particular source vertex $s$. Then we use these $\tau$ values to recursively calculate the $\delta_{s_k}(v)$, $\forall\ v, k$. The first stage uses breadth-first search to traverse the graph. In the second stage, we traverse the graph in reverse order from which it was explored during the search stage. We repeat this for each source vertex $s$ and sum the $\delta$ values for each vertex $v$ to obtain the $BC_k$ value.

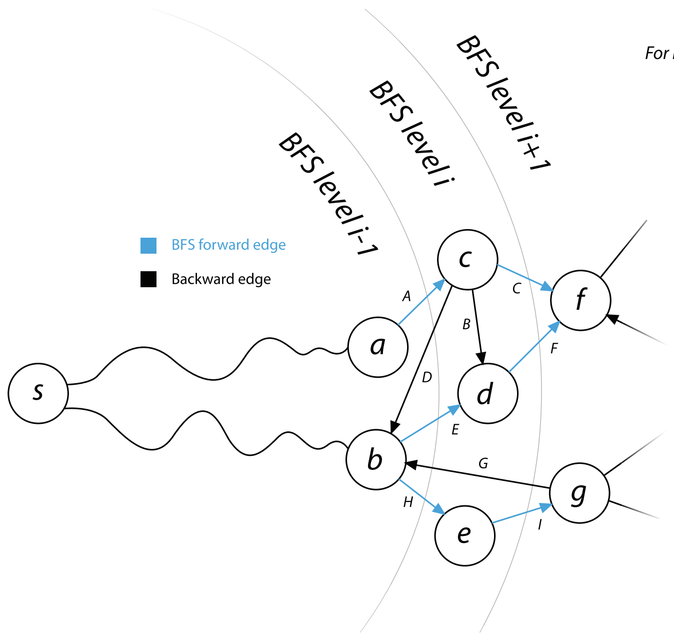
Fig. 10. Part III - Dependency accumulation by back-propagation

```
1: $phase \leftarrow phase - 1$
2: $\delta[i][t] \leftarrow 0\ \forall\ t \in V, 0 \leq i \leq k$
3: for $0 \leq k' \leq k$ do
4:   $p \leftarrow phase$
5:   while $p > 0$ do
6:     for all $v \in S[p]$ in parallel do
7:       for $0 \leq d \leq k'$ in parallel do
8:         for all $w \in Succ[d][v]$ in parallel do
9:           for $0 \leq i \leq k' - d$ do
10:            $sum \leftarrow 0$
11:            $e \leftarrow k' - d - i$
12:            for $0 \leq j \leq e$ do
13:              $sum \leftarrow sum + W(e-j, e, w, \tau) * \tau[j][v]$
14:            $\delta[k'][v] \leftarrow \delta[k'][v] + sum * \frac{\delta[i][w]}{\tau[0][w]^{e+1}}$
15:          $\delta[k'][v] \leftarrow \delta[k'][v] + \frac{\tau[k'-d][v]}{\Sigma_i \tau[i][w]}$
16:      $kBC[v] \leftarrow kBC[v] + \delta[k'][v]$
17:    $p \leftarrow p - 1$
```

We must modify the graph traversal phase from our previous work [12] in order to correctly propagate values of $\tau_{st_k}$. When $k = 0$, a single breadth-first is needed to propagate the value of $\tau$ from one level to the next. For $k > 0$, we must do $k + 1$ breadth-first searches. Notice that when we are updating $\tau$ values, for neighbors on the next level in the breadth-first search, we have $\tau_{sw_i} = \tau_{sv_i}$, $\forall\ i$, based on our generalized Bellman criterion. We must also increment the $\tau$ values for neighbors in the current or previous levels of the search. This may be challenging, since these neighbors then also need to pass on these values to their neighbors.

We solve this issue by only passing on different levels of $\tau$ to forward vertices and to those that are behind the breadth-first search frontier. Specifically, the forward propagation always trails the backward propagation by one level. For example, in the first step, we calculate and forward-propagate $\tau_0$ and back-propagate $\tau_1$. See Figure 11 for an illustration of this process.

We *back-propagate* $\tau_k$ to neighbors of the current vertex with appropriate $\Delta D$ and that have already been explored. We *forward-propagate* $\tau_k$ to neighboring vertices on the shortest path that we discover during the breadth-first search. During the first breadth-first search we store $k+1$ successor (or child) arrays. When we find a neighbor during breadth-first search whose



*For k=2:*

| *Edge* | ΔD | *σ–propogation* |
|---|---|---|
| *A* | 0 | σ[0][c] += σ[0][a] |
| | | σ[1][c] += σ[1][a] |
| | | σ[2][c] += σ[2][a] |
| *B* | 1 | σ[1][d] += σ[0][c] |
| | | σ[2][d] += σ[1][c] |
| *C* | 0 | σ[0][f] += σ[0][c] |
| | | σ[1][f] += σ[1][c] |
| | | σ[2][f] += σ[2][c] |
| *D* | 2 | σ[2][b] += σ[0][g] |
| *E* | 0 | σ[0][d] += σ[0][b] |
| | | σ[1][d] += σ[1][b] |
| | | σ[2][d] += σ[2][b] |
| *F* | 0 | σ[0][f] += σ[0][d] |
| | | σ[1][f] += σ[1][d] |
| | | σ[2][f] += σ[2][d] |
| *G* | 3 | |
| *H* | 0 | σ[0][e] += σ[0][b] |
| | | σ[1][e] += σ[1][b] |
| | | σ[2][e] += σ[2][b] |
| *I* | 0 | σ[0][g] += σ[0][e] |
| | | σ[1][g] += σ[1][e] |
| | | σ[2][g] += σ[2][e] |

🟦 Stage 1
🟩 Stage 2
⬛ Stage 3

Fig. 11. Illustration of $\tau$ propagation in the $k = 2$ case. Shown is a segment of the breadth-first search. The table represents the $\Delta D$ value as well as the propagation occurring as the result of each edge. The color of the $\tau$ propagation represents in which stage that addition will occur.

$\Delta D \leq k$, we append that neighbor's index to the $\Delta D^{th}$ successor array.

We need not re-run the breadth-first search after the first time; in a level-synchronous fashion, we may directly scan the successor arrays to perform our propagation, avoiding contention on a common vertex queue. We exploit parallelism in the traversal by exploring the neighbors of the current level of vertices concurrently. For a small-world graph, where graph diameter is small, the number of levels in the breadth-first search is correspondingly small, and parallelism is high. In the first traversal, all the vertices must add newly discovered vertices to an atomically accessed queue, which is the main bottleneck in the search. Since that work is done in the first phase, subsequent searches do not rely on this queue and avoid the sequential bottleneck to their parallelism.

For the $\delta-$accumulation step, we start by performing shortest-path accumulation as in Brandes's original algorithm. The $\delta_0$ values are used in the backward traversal to recursively calculate $\delta_1$, and so on and so forth.

### 6.3 Computing $k$-Betweenness on the Cray XMT

The Cray XMT implementation is similar to that used in previous work [12]. The successor arrays allows us to update our $\delta-$values without locking. The optimizations in our code for $k-$betweenness centrality exploit the fact that we are mostly interested in small values of $k$. In Figure 10, there are several nested loops, however most of them are very simple for small $k$ and unfurl quickly. By manually coding these loops for smaller values of $k'$, we significantly reduce the execution time since the time to set up and iterate over the small number of loop iterations quickly outstrips the actual useful work inside of them. For a SCALE 20 RMAT (Recursive MATrix graph generator [24]) graph (having $2^{20}$ vertices and $2^{23}$ edges), the time to compute 1-betweenness drops by a factor of two with this optimization.

In addition to optimizing for lower loop iterations, other considerations were taken for this architecture. Initially, temporary arrays were kept to store the number of children accumulated for a particular vertex during graph traversal. These temporary arrays required concurrent dynamic memory allocation. We were able to eliminate dynamic memory allocation altogether by accessing the source arrays directly, at the cost of addressing the larger array repeatedly. Reorganizing memory accesses to avoid dynamic allocation within the loop reduced runtime by more than 75%. Since the system has a large memory, we are encouraged to utilize extra memory in lieu of performing extra calculations (as long as we have sufficient network bandwidth): the expression $\Sigma_i \tau[i][w]$ in Figure 10 is precomputed and stored in an array for all values of $w$.

In Figure 12 we show the parallel scaling of our optimized code on the 128-processor Cray XMT. We reduced the execution time from several hours down to a few minutes for this problem. To accommodate the more than 12,000 hardware thread contexts available on this machine, we run multiple breadth-first searches in parallel and instruct the compiler to schedule main loop iterations using *loop futures*. This is necessary since each breadth-first search is

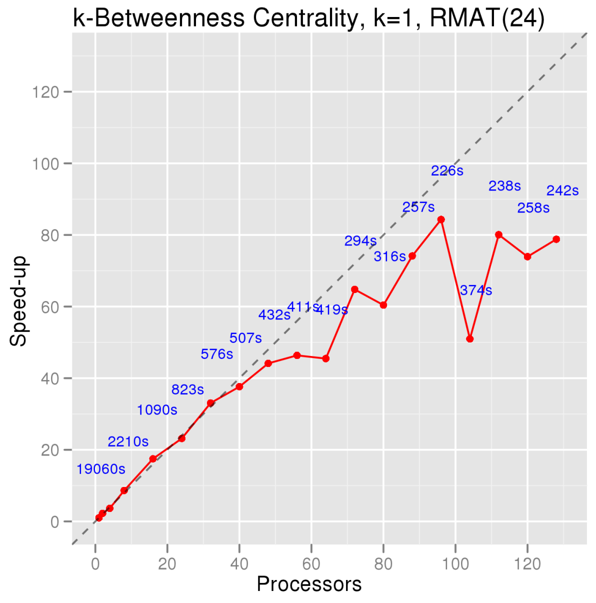


Fig. 12. Parallel scaling on the Cray XMT, for an RMAT generated undirected graph with approximately 16 million vertices and 135 million edges. Scaling is nearly linear up to 96 processors and speedup is roughly 78 on all 128 processors. $k = 1$ with 256 random sources (single node time 318 minutes). Execution times in seconds shown in blue.

dependent upon a single vertex queue that is accessed atomically and quickly becomes a hotspot. By doing so, however, the memory footprint is multiplied by the number of concurrent searches. On 128 processors, a graph with 135 million edges takes about 226 seconds to run for $k = 1$ *approximate* betweenness. This approximation is based on selecting a random sample of source vertices $s$. For these experiments, the number of starting vertices is 256. The plot shows good scaling up to our machine size.

### 6.4 Evaluating $k$-Betweenness

In order to explore the effect of various values of $k$ on the calculation of $k$-betweenness centrality, we apply our Cray XMT implementation to the ND-www graph data set [31]. This graph represents the hyperlink connections of web pages on the Internet. It is a directed graph with 325,729 vertices and 1,497,135 edges. Its structure demonstrates a power-law distribution in the number of neighbors. The graph displays characteristics typical of scale-free graphs found in social networks, biological networks, and computer networks.

To examine the graph data, we ran $k$-betweenness centrality for $k$ from 0 (traditional betweenness centrality) to 2. The betweenness scores are compared for each value of $k$. An analysis directly follows.

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

Fig. 13. The number of vertices ranked in selected percentiles for $k = 1$ and $k = 2$ whose betweenness centrality score was 0 for $k = 0$ (traditional BC). There were 14,320 vertices whose traditional BC score was 0, but whose $BC_k$ score for $k = 1$ was greater than 0. The ND-www graph contains 325,729 vertices and 1,497,135 edges.

Looking at the highest ranking vertices going from $k = 0$ to $k = 2$, the subset of vertices and the relative rankings change little. This seems to indicate that the paths $k$ longer than the shortest path lie along the same vertices as the shortest paths in this graph. As predicted, the traditional betweenness centrality metric fails to capture all of the information in the graph. When examining the $BC_k$ score for $k > 0$ of vertices whose score for $k = 0$ was 0 (no shortest paths pass through these vertices), it is clear that a number of very important vertices in the graph are not counted in traditional betweenness centrality. For $k = 1$, 417 vertices are ranked in the top 10 percent, but received a score of 0 for $k = 0$. In the 99th percentile are 11 vertices. Likewise, 12 vertices received a traditional BC score of 0, but ranked in the top 1 percent for $k = 2$. Figure 14 shows 14,320 vertices whose betweenness centrality score for $k = 0$ was 0, but had a $k$-betweenness centrality score of greater than 0 for $k = 1$.

Given that the execution time of this algorithm grows exponentially with $k$, it is desirable to understand the effect of choosing a given value for $k$. Figure 15 shows that increasing $k$ from 0 to 1 captures significantly more path data. However, increasing from $k = 1$ to $k = 2$ displays much less change. It is reasonable to believe that small values of $k$ for some applications may capture an adequate amount of information while remaining computable.

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

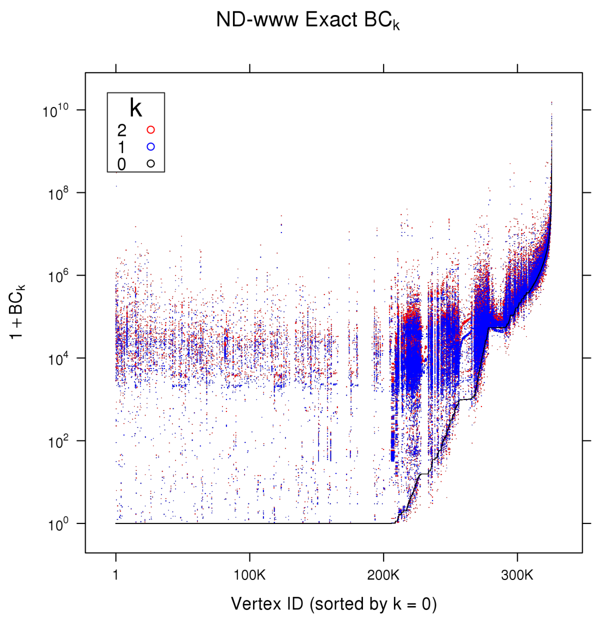


Fig. 14. Per-vertex betweenness centrality scores for $k = 0$ (in black), 1 (in blue), and 2 (in red), sorted in ascending order for $k = 0$ for the Notre Dame web crawl dataset [31]. Note the number of vertices whose score is several orders of magnitude larger for $k = 1$ or $2$ than for traditional betweenness centrality.

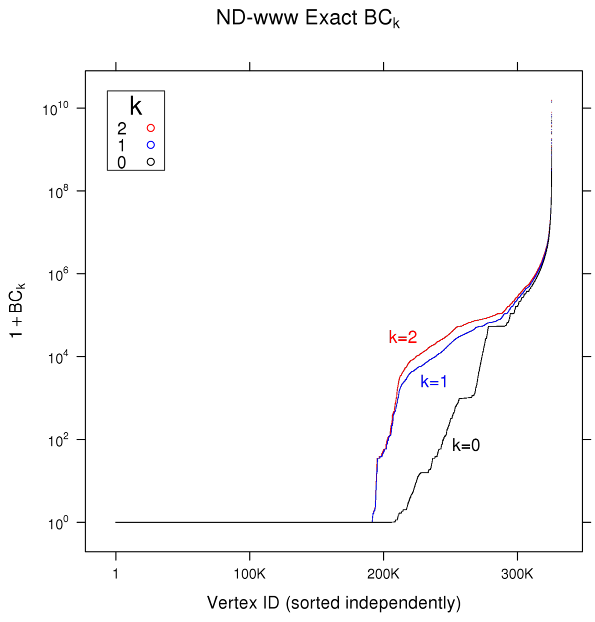


Fig. 15. Per-vertex betweenness centrality scores for $k = 0$, 1, and 2, sorted independently in ascending order for each value of $k$.

## 7 Conclusion

The explosion of semantic data in digital form has necessitated the development of algorithms and software to gain even a first-order understanding of the relationships at work. The computational and storage requirements of large datasets has brought about parallel, multithreaded supercomputers like the Cray XMT. GraphCT incorporates multithreaded implementations of cutting edge algorithms and traditional analytics. Running on the Cray XMT, we are able to produce a summary analysis of an unknown input graph with billions of vertices and edges in several minutes to several hours; a task that would be impossible on conventional multicore servers and workstations.

The processing framework used in GraphCT enables a series of complex analytics to run with the option of passing the results of one to the input of the next. A researcher with an unknown data source is able to prepare a custom workflow of routines that will produce a report on the graph characteristics. Our tool has been used successfully in prior work [15] to uncover hidden relationships in online social networks.

As the quantity and richness of data continues to grow, algorithms must continue to evolve to handle the scalability and complexity challenges. Now that we are able to glean a first-order understanding our input data, the challenge will be to engineer algorithms and applications that can quickly reveal the important structures and trends in today's networks.

## Acknowledgments

This work was supported in part by the Pacific Northwest National Lab (PNNL) Center for Adaptive Supercomputing Software for MultiThreaded Architectures (CASS-MT). We thank PNNL and Cray for providing access to Cray XMT systems.

## References

[1] Facebook, "User statistics," March 2012, http://newsroom.fb.com/content/default.aspx?NewsAreaId=22.

[2] C. L. Borgman, J. C. Wallis, M. S. Mayernik, and A. Pepe, "Drowning in data: digital library architecture to support scientific use of embedded sensor networks," in *Proceedings of the 7th ACM/IEEE-CS joint conference on Digital libraries*, ser. JCDL '07, 2007, pp. 269–277.

[3] F. Liljeros, C. Edling, L. Amaral, H. Stanley, and Y. Åberg, "The web of human sexual contacts," *Nature*, vol. 411, pp. 907–908, 2001.

[4] H. Jeong, S. Mason, A.-L. Barabási, and Z. Oltvai, "Lethality and centrality in protein networks," *Nature*, vol. 411, pp. 41–42, 2001.

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

[6] R. Kouzes, G. Anderson, S. Elbert, I. Gorton, and D. Gracio, "The changing paradigm of data-intensive computing," *Computer*, vol. 42, no. 1, pp. 26–34, jan. 2009.

[7] V. Batagelj and A. Mrvar, "Pajek - program for large network analysis," *Connections*, vol. 21, pp. 47–57, 1998.

[8] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. Complex Systems, p. 1695, 2006. [Online]. Available: http://igraph.sf.net

[9] D. Auber, "Tulip," in *9th Symp. Graph Drawing*, ser. Lecture Notes in Computer Science, P. Mutzel, M. Jünger, and S. Leipert, Eds., vol. 2265. Springer-Verlag, 2001, pp. 335–337.

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

[11] D. Watts and S. Strogatz, "Collective dynamics of small world networks," *Nature*, vol. 393, pp. 440–442, 1998.

[12] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarría-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *Proc. Workshop on Multithreaded Architectures and Applications (MTAAP'09)*, Rome, Italy, May 2009.

[13] K. Jiang, D. Ediger, and D. A. Bader, "Generalizing $k$-Betweenness centrality using short paths and a parallel multithreaded implementation," in *The 38th International Conference on Parallel Processing (ICPP 2009)*, Vienna, Austria, Sep. 2009.

[14] D. Ediger, K. Jiang, J. Riedy, and D. A. Bader, "Massive streaming data analytics: A case study with clustering coefficients," in *Workshop on Multithreaded Architectures and Applications (MTAAP)*, Atlanta, Georgia, Apr. 2010.

[15] D. Ediger, K. Jiang, J. Riedy, D. A. Bader, C. Corley, R. Farber, and W. N. Reynolds, "Massive social network analysis: Mining twitter for social good," *Parallel Processing, International Conference on*, pp. 583–593, 2010.

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

[17] D. Gregor and A. Lumsdaine, "Lifting sequential graph algorithms for distributed-memory parallel computation," *SIGPLAN Not.*, vol. 40, no. 10, pp. 423–437, 2005.

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

[19] J. Berry, B. Hendrickson, S. Kahan, and P. Konecny, "Software and algorithms for graph queries on multithreaded architectures," in *Proc. Workshop on Multithreaded Architectures and Applications*, Long Beach, CA, March 2007.

[20] K. Lang, "Finding good nearly balanced cuts in power law graphs," Yahoo! Research, Tech. Rep., 2004.

[21] M. E. J. Newman, "Modularity and community structure in networks," *Proceedings of the National Academy of Sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.

[22] B. Bollobas, *Modern Graph Theory*. Springer, 1998.

[23] Y. Shiloach and U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm," *J. Algs.*, vol. 3, no. 1, pp. 57–67, 1982.

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

[25] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

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

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

[28] R. Guimerà, S. Mossa, A. Turtschi, and L. Amaral, "The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles," *Proceedings of the National Academy of Sciences USA*, vol. 102, no. 22, pp. 7794–7799, 2005.

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

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

[31] A.-L. Barabási, "Network databases," 2007, http://www.nd.edu/~networks/resources.htm.

**David Ediger** is a Ph.D. candidate in Electrical and Computer Engineering and a research assistant in the High Performance Computing Lab at Georgia Tech. He completed his B.S. in Computer Engineering in 2008 under the direction of Dr. Tarek El-Ghazawi at The George Washington University focusing on Unified Parallel C (UPC) and reconfigurable systems and an M.S. in Electrical and Computer Engineering from Georgia Tech in 2010. David is principal developer of GraphCT, the graph characterization package for highly parallel, massive graph analysis.

**Karl Jiang** is a Ph.D. candidate in Computer Science at Georgia Institute of Technology. He received his B. S. in Computer Engineering from the University of Miami in 2007. He has done work in massively parallel bioinformatics and graph analysis, including contributions to MPI BLAST and HMMER.

**E. Jason Riedy** is a Research Scientist II in the School of Computational Science and Engineering at Georgia Tech. He is developing a software framework for parallel analysis of massive, streaming graph data (STING). He has codes extending and included in the widely used packages LAPACK and the extended-precision BLAS (XBLAS). He has contributed to GNU Octave, GNU Emacs, GNU R packages, git, and other free software. He was a member of the IEEE 754 revision committee and has presented widely on software development and research. His Ph.D. is from the University of California, Berkeley in 2010 under Dr. James Demmel on sparse linear algebra and parallel graph optimization. His dual B.S. in Computer Science and Mathematics is from the University of Florida in 1998.

**David A. Bader** is a Full Professor in the School of Computational Science and Engineering, College of Computing, at Georgia Institute of Technology, and Executive Director for High Performance Computing. Dr. Bader is a lead scientist in the DARPA Ubiquitous High Performance Computing (UHPC) program. He received his Ph.D. in 1996 from The University of Maryland, and his research is supported through highly-competitive research awards, primarily from NSF, NIH, DARPA, and DOE. Dr. Bader serves on the Research Advisory Council for Internet2, the Steering Committees of the IPDPS and HiPC conferences, the General Chair of IPDPS 2010 and Chair of SIAM PP12. He is an associate editor for several high impact publications including the Journal of Parallel and Distributed Computing (JPDC), ACM Journal of Experimental Algorithmics (JEA), IEEE DSOnline, Parallel Computing, and Journal of Computational Science, and has been an associate editor for the IEEE Transactions on Parallel and Distributed Systems (TPDS). Dr. Bader's interests are at the intersection of high-performance computing and real-world applications, including computational biology and genomics and massive-scale data analytics. He has co-chaired a series of meetings, the IEEE International Workshop on High-Performance Computational Biology (HiCOMB), co-organized the NSF Workshop on Petascale Computing in the Biological Sciences, written several book chapters, and co-edited special issues of the Journal of Parallel and Distributed Computing (JPDC) and IEEE TPDS on high-performance computational biology. He is also a leading expert on multicore, manycore, and multithreaded computing for data-intensive applications such as those in massive-scale graph analytics. He has co-authored over 100 articles in peer-reviewed journals and conferences, and his main areas of research are in parallel algorithms, combinatorial optimization, massive-scale social networks, and computational biology and genomics. Prof. Bader is an IEEE Fellow, a National Science Foundation CAREER Award recipient, and has received numerous industrial awards from IBM, NVIDIA, Intel, Sun Microsystems, and Microsoft Research. He served as a member of the IBM PERCS team for the DARPA High Productivity Computing Systems program, was a distinguished speaker in the IEEE Computer Society Distinguished Visitors Program, and has also served as Director of the Sony-Toshiba-IBM Center of Competence for the Cell Broadband Engine Processor.