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 data sets, 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 data sets. On a synthetic power law graph with 2 billion vertices and 17 billion edges, we can find the connected components in 2 minutes. We can estimate the betweenness centrality of a similar graph with 537 million vertices and over 8 billion edges in under 1 hour. GraphCT is built for portability and performance.

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. To analyze today's graphs and the semantic data of the future, scalable algorithms and machine architectures are needed for data-intensive computing. GraphCT [11] is a collection of new parallel and scalable algorithms for static graph analysis. These algorithms, running atop multithreaded architectures such as the Cray XMT, can analyze graphs with hundreds of millions of vertices and billions of edges in minutes, instead of days or weeks. GraphCT is able to, for the first time, enable analysts and domain experts to conduct in-depth analytical workflows of their data at massive scales.

The foundation of GraphCT is a modular kernel-based design using efficient data representations in which an analysis workflow can be expressed through a series of function calls. Fig. 1 illustrates the high-level framework. All functions are required to use a single graph data representation. The use of a single common data structure enables plug-and-play capability as well as ease of implementation and 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. Fig. 2 illustrates an example workflow. Analysis can be conducted on unweighted and weighted graphs, undirected and directed. Limited sections of GraphCT are parallelized for parallel platforms other than the Cray XMT.

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 (CSR) 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" [12] 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

- *The authors are with the School of Computational Science and Engineering, Georgia Institute of Technology, Klaus Advanced Computing Building, 266 Ferst Drive, Atlanta, GA 30332-0765. E-mail: dediger@gatech.edu, findingthenumber@gmail.com, {jason.riedy, bader}@cc.gatech.edu.*

Manuscript received 17 Apr. 2012; revised 11 Oct. 2012; accepted 31 Oct. 2012; published online 28 Nov. 2012.
Recommended for acceptance by I. Ahmad.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2012-04-0393.
Digital Object Identifier no. 10.1109/TPDS.2012.323.

1045-9219/13/$31.00 © 2013 IEEE Published by the IEEE Computer Society

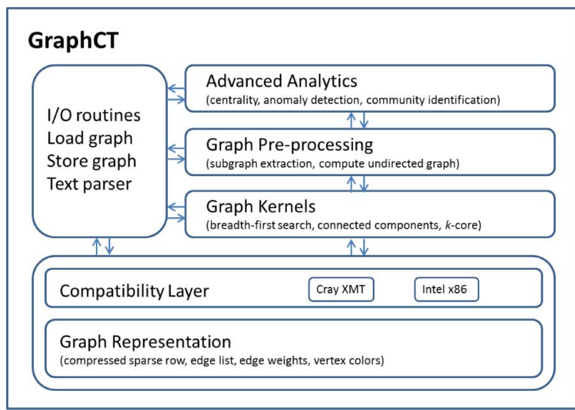


Fig. 1. GraphCT is an open-source framework for developing scalable multithreaded graph analytics in a cross-platform environment.

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 a search 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 [13]. Lightweight and fine-grained synchronization enable algorithm designers to expose large quantities of parallelism in the application.

In our previous work [14], [15], [16], [17], 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 data sets 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 as follows:

- 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.
- k-Betweenness Centrality, a new parallel algorithm for finding important vertices in a network that is robust to edge deletions.

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 and 5 will present the implementation and performance of connected components and clustering coefficients, respectively. The design and implementation of the k-betweenness centrality algorithm is described in the supplemental section, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.323.

2 Motivation and Requirements

A number of software applications have been developed for analyzing and visualizing graph data sets. 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 and do not take advantage of parallel systems. Pajek has been known to run complex graph analytics on inputs with up to two million vertices, but many other applications are limited to tens or hundreds of thousands of vertices.

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

The development of new scalable algorithms and frameworks for massive graph analysis is the subject of many research efforts. The Parallel Boost Graph Library (PBGL) [19] is a C++ library for distributed memory graph computations. The API is similar to that of the boost graph library. The authors report scalable performance up to about 100 processors. Distributed memory graph processing often requires partitioning and data replication, which can be challenging for some classes of graphs. Google's Pregel [18]

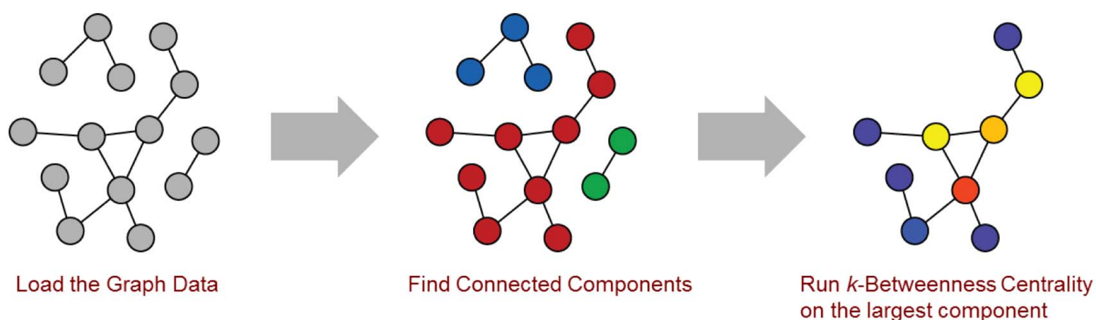


Fig. 2. An example user analysis workflow in which the graph is constructed, the vertices are labeled according to their connected components, and a single component is extracted for further analysis using several complex metrics, such as betweenness centrality.

TABLE 1
Graph Analysis Packages and Frameworks Currently under Active Development

Package	Interface	Parallel	Memory	O($Edges$)	Analytics	Frameworks
Pregel [18]	C++	X	Distributed on-disk	127 billion		X
MTGL [13]	C++	X	Shared (Cray XMT)	35 billion		X
GraphCT	C	X	Shared (Cray XMT)	17 billion	X	X
PBGL [19]	C++	X	Distributed in-memory	17 billion		X
KDT [20]	Python	X	Distributed in-memory	8 billion	X	X
Pegasus [21]	Hadoop	X	Distributed on-disk	6.6 billion		X
NetworkX [22]	Python		Shared	100 million	X	
SNAP [23]	C	X	Shared	32 million	X	
Pajek [7]	Windows		Shared	16 million	X	
igraph [8]	R		Shared	Millions	X	

uses a MapReduce-like programming model for describing vertex-centric graph computations on large, distributed memory clusters. Buluç and Madduri [24] have demonstrated high performance techniques for scaling breadth-first search on distributed memory supercomputers.

SNAP [23] is an open-source parallel library for network analysis and partitioning using multicore workstations. It is parallelized using OpenMP and provides a simple API with support for very large-scale graphs. It is one of the only libraries that provides a suite of algorithms for community detection. The Knowledge Discovery Toolbox (KDT) [20] enables high performance graph analysis in Python by leveraging highly tuned sparse linear algebra kernels.

Sandia's Multithreaded Graph Library (MTGL) [13] is a C++ library for implementing graph applications on multithreaded architectures, particularly the Cray XMT. MTGL uses the notion of a "visitor" class to describe operations that take place when a vertex is visited, such as during a breadth-first search.

Other approaches to large graph problems include the NoSQL graph databases used by the Semantic Web community. These graph databases implement RDF triple stores that support ACID properties. They lack relational database schemas, but include query languages such as SPARQL [25]. Also, WebGraph is a graph compression technique for large graphs generated through web crawls [26].

Given the immense size in memory of the graphs of interest, it is not possible to store a separate representation for each analysis kernel. Since the key capability of GraphCT is running a number of analytics against an unknown data source, we employ a simple, yet powerful framework for our computation. Each kernel implementation is required to use a common graph data structure. By using the same data structure for each kernel, all kernels can be run in succession (or even in parallel if resources allow) without the need to translate the graph between data structures. In the client/server model, a GraphCT server process loads the graph into memory and shares it in read-only mode with all client analytic processes, amortizing the time required to load the data and generate the graph. The results of one computation can easily influence the next computation, such as the extraction of one or more connected components for more in-depth study.

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

2.1 The Cray XMT

To facilitate scaling to the sizes of massive data sets previously described, GraphCT utilizes the massive shared memory and multithreading capabilities of the Cray XMT. Large planar graphs, such as road networks, can be partitioned with small separators and analyzed in distributed memory with good computation-to-communication ratios at the boundaries. Graphs arising from massive social networks, on the other hand, are challenging to partition and lack small separators [23], [27]. 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

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

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

cycle. Given sufficient parallelism and hardware contexts, the processor's execution units can stay busy and hide some or all of the memory latency.

Since a 128-processor Cray XMT contains about 12,000 user hardware contexts, it is the responsibility of the programmer to reveal a large degree of parallelism in the code. Coarse- as well as fine-grained parallelism can be exploited using Cray's parallelizing compiler. The programmer inserts `#pragma` statements to assert that a loop's iterations are independent (see Fig. 3). Often iterations of a loop will synchronize on shared data. To exploit this parallelism, the Cray XMT provides low-cost fine-grained synchronization primitives such as full-empty bit synchronization and atomic fetch-and-add [28]. 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.

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.) to perform a calculation, but the edge and vertex data should not be duplicated. This design principle allows for efficient use of the machine's memory to support massive graphs and complex queries. We refer the reader to [29] for an in-depth study of graph algorithms.

The data representation used internally for the graph is an extension based on CSR format. In CSR, contiguous edges originating from the same source vertex are stored by destination vertex only. An offset array indicates at which location a particular vertex's edges begin. The common access pattern is 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. 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 Client/Server Mode

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

3.3 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 [12]. For undirected graphs, we can compute the global clustering coefficient, which is a single number describing the entire graph, or the local clustering coefficients, which is a per-vertex measure of triangles. For directed graphs, several variations have been proposed, and we have adopted the transitivity coefficients, which is a natural extension of the local clustering coefficient.

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

3.4 Connected Components

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

In Section 4, we will offer in-depth coverage of the algorithm, implementation, and performance of our connected components routine. We use a shared memory version of the classical Shiloach and Vishkin algorithm. 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.

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

3.6 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 is helpful to estimate the rate of transmission and the time to full coverage. Calculating the diameter exactly requires an all-pairs shortest path computation, which is prohibitive for the large graphs of interest.

In GraphCT, we estimate the diameter by random sampling. Given a fixed number of source vertices (expressed as a percentage of the total number of vertices), a breadth-first search is executed from each chosen source. The length of the longest path found during that search is compared to the global maximum seen so far and updated if it is longer. With each sample, we more closely approximate the true diameter of the graph. Ignoring the existence of long chains of vertices, we can obtain a reasonable estimate with only a small fraction of the total number of breadth-first searches required to get an exact diameter [30]. However, GraphCT leaves the option of the number of samples to the user, so an exact computation can be requested.

Obtaining a reasonable estimate of the graph diameter can have practical consequences for the analysis kernels. A kernel running a level-synchronous breadth-first search will require a queue for each level. The total number of items in each 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 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.

3.7 Graph Parsing and Generation

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

GraphCT supports the common DIMACS format, 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.

For instances when real data are not available with the scale or characteristics of interest, GraphCT is able to provide graph generators that will provide an output file on disk that can be read in for kernel testing. GraphCT includes an implementation of the RMAT graph generator, which was used in the DARPA High Productivity Computing Systems (HPCS) Scalable Synthetic Compact Applications benchmark #2 (SSCA2). This generator uses repeated sampling from a Kronecker product to produce a random graph with a degree distribution similar to those arising from social networks. The generator takes as input the probabilities a, b, c, and d that 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.

3.8 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 [31].

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

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

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

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 [32]. The value of Φ ranges from 0 to 1. While the formula above is for an unweighted graph, it can be generalized to weighted networks by summing edge weights instead of degrees.

Given an edge cut expressed as a 2-coloring of vertices, the conductance is computed by iterating over all edges. Each edge is placed in one of three buckets: 1) both endpoints belong to the same partition, 2) the endpoints are in Partitions A and B, respectively, or 3) the endpoints are in Partitions 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.

3.9 Betweenness Centrality

Betweenness centrality has proved a useful analytic for ranking important vertices and edges in large graphs. Betweenness centrality is a measure of the number of shortest paths in a graph passing through a given vertex [33]. For a graph $G(V, E)$, let σ_{st} denote the number of shortest paths between vertices s and t, and $\sigma_{st}(v)$ the count of shortest paths that pass through a specified vertex v. The betweenness centrality of v is defined as

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

GraphCT on the 128-processor Cray XMT recorded 606 million traversed edges per second on a scale-free graph with 135 million vertices and 2.14 billion edges. KDT implements betweenness centrality on a distributed-memory cluster using Combinatorial BLAS. The authors demonstrate performance results on a scale-free graph with approximately 262,000 vertices and 4 million edges. They report 125 million traversed edges per second using 256 cores (24 cores per node) [20].

We present k-betweenness centrality, an extension of Freeman's betweenness centrality metric considering additional paths in the graph, in the online supplemental section. We give a parallel, multithreaded algorithm and compute it on the Cray XMT demonstrating scalable performance for massive graphs.

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. 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 to isolate those vertices of greatest interest.

The Shiloach and Vishkin algorithm [34] 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.

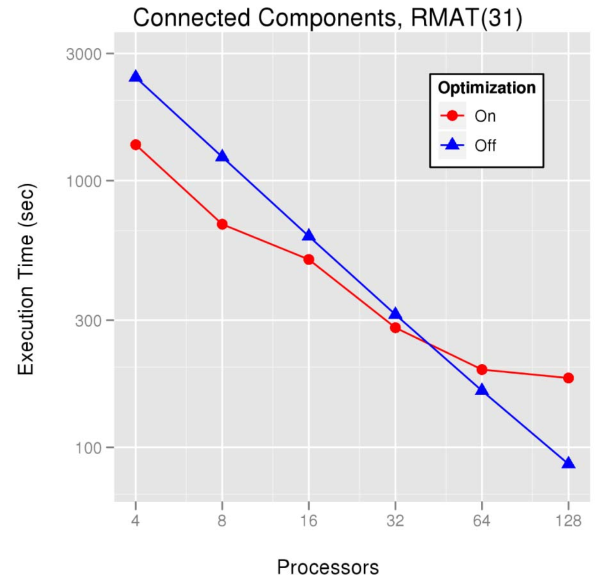


Fig. 4. Scalability of Shiloach-Vishkin connected components with and without tree-climbing optimization on the Cray XMT. The input graph is an RMAT generated graph with approximately 2 billion vertices and 17 billion edges. The speedup is 113× on 128 processors.

In Algorithm 1, each vertex is initialized to its own unique color (Line 3). At each step, neighboring vertices greedily color each other such that the vertex with the lowest ID wins (Lines 7 and 8). The process ends when each vertex is the same color as its neighbors. The number of steps is proportional to the diameter of the graph, so for small-world networks the algorithm converges quickly.

Algorithm 1. Parallel multithreaded version of Shiloach-Vishkin algorithm for finding the connected components of a graph.
Input: $G(V, E)$
Output: $M[1..n]$, where $M[v]$ is the component to which vertex v belongs

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

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

In Lines 10 through 12 of Algorithm 1, each vertex climbs the component tree, relabeling itself. This optimization can reduce the number of iterations required. In Fig. 4, we plot the execution time for an RMAT graph with 2 billion vertices and 17 billion edges as a function of the number of Cray XMT processors. At low processor counts, the optimization

TABLE 2
Running Times in Seconds for Connected Components on a 128-Processor Cray XMT

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

on Lines 10 through 12 shortens the execution time. However, it creates a memory hotspot and the additional contention at large processor counts produces a less scalable implementation. Removing Lines 10 through 12 from the algorithm results in a 113× speedup on 128 processors.

In Table 2, 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 processors, we can determine the connected components of the graph in 2 minutes.

In [35], MTGL running on a 128-processor Cray XMT computes the connected components of an RMAT graph (with so-called "nasty" parameters) with 33.5 million vertices and an average degree of 8 in approximately 8 seconds. On a generated graph with the same RMAT parameters on the same size machine, GraphCT is able to compute the connected components in 1.89 seconds.

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 [12]. We adopt the terminology of Watts and Strogatz [12] 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 Fig. 5. A closed triplet is defined as three vertices in which there are three edges, or Fig. 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

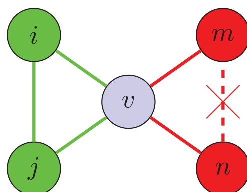


Fig. 5. There are two triplets around v in this undirected graph. The triplet (m, v, n) is open, there is no edge $\langle m, n\rangle$. The triplet (i, v, j) is closed.

$$C = \frac{\text{number of closed triplets}}{\text{number of triplets}}. \tag{4}$$

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

$$C_v = \frac{\text{number of closed triplets around } v}{\text{number of triplets around } v}. \tag{5}$$

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

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

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

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 presorted. The multithreaded implementation also is straightforward; we parallelize over the vertices.

Fig. 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 calculate all clustering coefficients in 87 minutes on a single processor and 56 seconds on 128 processors. The speedup is 94×. 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, and 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 algorithm scales linearly to 128 processors. The total execution time is about 27 seconds.

These results highlight the challenges of developing scalable algorithms on massive graphs. Where commodity platforms often struggle to obtain a speedup, the latency tolerance and massive multithreading of the Cray XMT enables linear scalability on regular, uniform graphs. The discrepancy between the RMAT scalability (left) and the road network (right) is an artifact of the power law degree distribution of the former. Despite the complex and

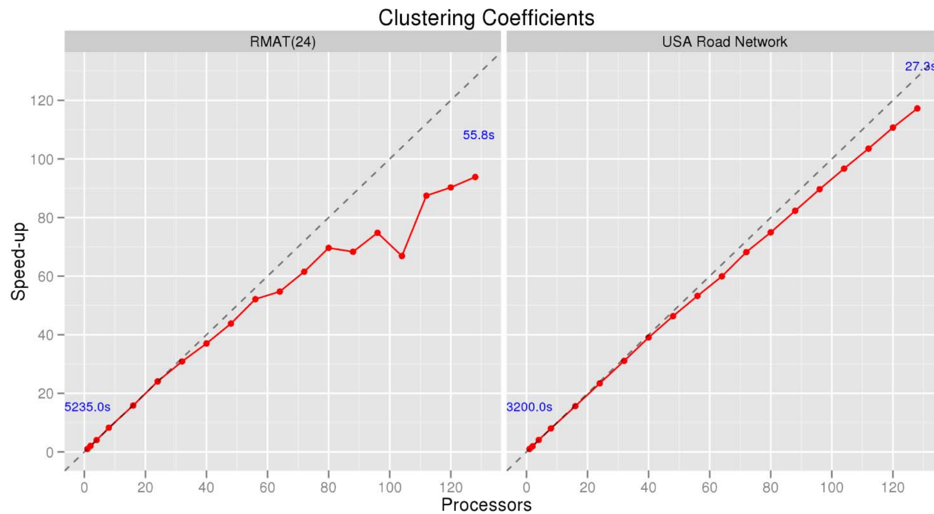


Fig. 6. Scalability of the local clustering coefficients kernel on the Cray XMT. On the left, the input graph is an undirected RMAT generated graph with approximately 16 million vertices and 135 million edges. The speedup is 94× on 128 processors. On the right, the input graph is the USA road network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

irregular graph topology, GraphCT is still able to scale up to 128 processors.

There are several variations of clustering coefficients for directed graphs. A straightforward approach is to apply the definition directly and count the number of triangles, where a triangle now requires six edges instead of three. A more sophisticated approach is called the transitivity coefficients. Transitivity coefficients count the number of transitive triplets in the numerator. A transitive triplet is one in which edges exist from vertex a to vertex b and from vertex b to vertex c, with a shortcut edge from vertex a to vertex c [36].

Fig. 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 speedup is 90×.

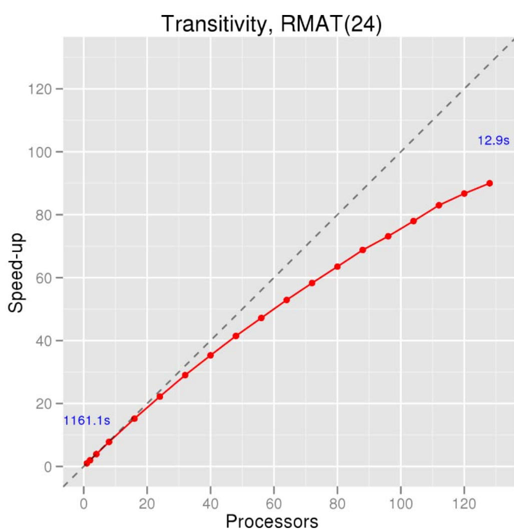


Fig. 7. Scalability of the transitivity coefficients kernel on the Cray XMT. The input graph is a directed RMAT generated graph with approximately 16 million vertices and 135 million edges. Execution times in seconds are shown in blue. On 128 processors, we achieve a speedup of 90×.

6 Conclusions

The computational and storage requirements of large data sets have brought about parallel, multithreaded supercomputers like the Cray XMT. GraphCT leverages multithreaded implementations of cutting edge graph algorithms and traditional analytics. Running on the 128-processor Cray XMT with 1-TB main memory, GraphCT calculates the local clustering coefficients of a scale-free graph with 135 million edges in one minute. The connected components of a scale-free graph with 4.27 billion edges are determined in 2 minutes. The betweenness centrality of a scale-free graph with 2.14 billion edges can be approximated (using 256 randomly sampled vertices) in 15 minutes. These algorithms exploit fine-grained parallelism and low-overhead synchronization to overcome the challenges of workload imbalance and hotspotting that are inherent in scale-free graph analytics.

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 set 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 [17] to uncover hidden relationships in Twitter.

GraphCT has the capability to run on billion-scale graphs with better performance than leading distributed memory and MapReduce-like frameworks. Unlike these frameworks, GraphCT is an end-to-end graph characterization toolkit that provides high performance, multithreaded graph analysis algorithms as well as a C language environment for working with graph data.

Acknowledgments

This work was supported in part by the Pacific Northwest National Laboratory (PNNL), Center for Adaptive Supercomputing Software for MultiThreaded Architectures. The authors thank PNNL and Cray for providing access to Cray XMT systems.

REFERENCES

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

[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," *Proc. Seventh ACM/IEEE-CS Joint Conf. Digital Libraries,* pp. 269-277, 2007.

[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," *Physical 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," *InterJ.,* vol. Complex Systems, p. 1695, http://igraph.sf.net, 2006.

[9] D. Auber, "Tulip," *Proc. Ninth Symp. Graph Drawing,* P. Mutzel, M. Jünger, and S. Leipert, eds., pp. 335-337, 2001.

[10] *Analytic Technologies,* "UCINET 6 Social Network Analysis Software," http://www.analytictech.com/ucinet.htm, 2013.

[11] "GraphCT," ver. 0.8.0, http://www.cc.gatech.edu/bader, Sept. 2012.

[12] D. Watts and S. Strogatz, "Collective Dynamics of Small World Networks," *Nature,* vol. 393, pp. 440-442, 1998.

[13] J. Berry, B. Hendrickson, S. Kahan, and P. Konecny, "Software and Algorithms for Graph Queries on Multithreaded Architectures," *Proc. Workshop Multithreaded Architectures and Applications,* Mar. 2007.

[14] 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 Data Sets," *Proc. Workshop Multithreaded Architectures and Applications (MTAAP '09),* May 2009.

[15] K. Jiang, D. Ediger, and D.A. Bader, "Generalizing k-Betweenness Centrality Using Short Paths and a Parallel Multithreaded Implementation," *Proc. 38th Int'l Conf. Parallel Processing (ICPP '09),* Sept. 2009.

[16] D. Ediger, K. Jiang, J. Riedy, and D.A. Bader, "Massive Streaming Data Analytics: A Case Study with Clustering Coefficients," *Proc. Workshop Multithreaded Architectures and Applications (MTAAP),* Apr. 2010.

[17] 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," *Proc. Int'l Conf. Parallel Processing,* pp. 583-593, 2010.

[18] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," *Proc. Int'l Conf. Management of Data,* pp. 135-146, http://doi.acm.org/10.1145/1807167.1807184, 2010.

[19] D. Gregor and A. Lumsdaine, "Lifting Sequential Graph Algorithms for Distributed-Memory Parallel Computation," *ACM SIGPLAN Notices,* vol. 40, no. 10, pp. 423-437, 2005.

[20] A. Lugowski, D. Alber, A. Buluç, J.R. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis, "A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis," *Proc. 12th SIAM Int'l Conf. Data Mining (SDM '12),* pp. 930-941, http://siam.omnibooksonline.com/2012datamining/data/papers/158.pdf/, Apr. 2012.

[21] U. Kang, B. Meeder, and C. Faloutsos, "Spectral Analysis for Billion-Scale Graphs: Discoveries and Implementation," *Proc. 15th Pacific-Asia Conf. Advances in Knowledge Discovery and Data Mining,* pp. 13-25, http://dl.acm.org/citation.cfm?id=2022850.2022852, 2011.

[22] A.A. Hagberg, D.A. Schult, and P.J. Swart, "Exploring Network Structure, Dynamics, and Function Using NetworkX," *Proc. Seventh Python in Science Conf. (SciPy '08),* pp. 11-15, Aug. 2008.

[23] 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," *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS '08),* Apr. 2008.

[24] A. Buluç and K. Madduri, "Parallel Breadth-First Search on Distributed Memory Systems," *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis,* pp. 65:1-65:12, http://doi.acm.org/10.1145/2063384.2063471, 2011.

[25] E. Prud'hommeaux and A. Seaborne, "SPARQL Query Language for RDF," technical report, W3C, http://www.w3.org/TR/rdf-sparql-query/, 2006.

[26] P. Boldi and S. Vigna, "The Webgraph Framework I: Compression Techniques," *Proc. 13th Int'l Conf. World Wide Web,* pp. 595-602, http://doi.acm.org/10.1145/988672.988752, 2004.

[27] K. Lang, "Finding Good Nearly Balanced Cuts in Power Law Graphs," technical report, Yahoo! Research, 2004.

[28] J. Feo, D. Harper, S. Kahan, and P. Konecny, "Eldorado," *Proc. Second Conf. Computing Frontiers,* pp. 28-34, http://doi.acm.org/10.1145/1062261.1062268, 2005.

[29] S. Even, *Graph Algorithms.* W.H. Freeman & Co., 1979.

[30] D. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating Betweenness Centrality," *Proc. Fifth Workshop Algorithms and Models for the Web-Graph (WAW '07),* pp. 134-137, Dec. 2007.

[31] M.E.J. Newman, "Modularity and Community Structure in Networks," *Proc. Nat'l Academy of Sciences,* vol. 103, no. 23, pp. 8577-8582, 2006.

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

[33] L. Freeman, "A Set of Measures of Centrality Based on Betweenness," *Sociometry,* vol. 40, no. 1, pp. 35-41, 1977.

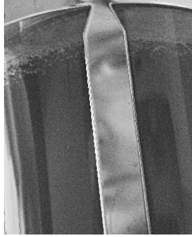
[34] Y. Shiloach and U. Vishkin, "An $O(\log n)$ Parallel Connectivity Algorithm," *J. Algorithms,* vol. 3, no. 1, pp. 57-67, 1982.

[35] S.J. Plimpton and K.D. Devine, "MapReduce in MPI for Large-Scale Graph Algorithms," *Parallel Computing,* vol. 37, no. 9, pp. 610-632, http://dx.doi.org/10.1016/j.parco.2011.02.004, Sept. 2011.

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

David Ediger received the BS degree in computer engineering in 2008 under the direction of Dr. Tarek El-Ghazawi at the George Washington University focusing on Unified Parallel C and reconfigurable systems and the MS degree in electrical and computer engineering from Georgia Tech in 2010. He is currently working toward the PhD degree in electrical and computer engineering. He is a research assistant in the High Performance Computing Lab, Georgia Tech. He is a principal developer of GraphCT, the graph characterization package for highly parallel, massive graph analysis. He is a member of the IEEE.

Karl Jiang received the BS degree in computer engineering from the University of Miami in 2007. He is currently working toward the PhD degree in computer science at the Georgia Institute of Technology. He has done work in massively parallel bioinformatics and graph analysis, including contributions to MPI BLAST and HMMER.



E. Jason Riedy received the dual BS degree in computer science and mathematics from the University of Florida in 1998. He received the PhD degree from the University of California, Berkeley, in 2010 under Dr. James Demmel on sparse linear algebra and parallel graph optimization. He is a research scientist II at the School of Computational Science and Engineering, 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.

David A. Bader received the PhD degree from the University of Maryland in 1996, and his research was supported through highly competitive research awards, primarily from the US National Science Foundation (NSF), NIH, US Defense Advanced Research Projects Agency (DARPA), and US Department of Energy. He is a full professor at the School of Computational Science and Engineering, College of Computing, Georgia Institute of Technology, and executive director for High Performance Computing. He is a lead scientist in the DARPA Ubiquitous High Performance Computing program. He 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*, *ACM Journal of Experimental Algorithmics*, *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)*. His research 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 cochaired a series of meetings, the IEEE International Workshop on High-Performance Computational Biology, co-organized the NSF Workshop on Petascale Computing in the Biological Sciences, written several book chapters, and coedited special issues of the *Journal of Parallel and Distributed Computing* 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 coauthored more than 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. He is a fellow of the IEEE, the US 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.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.