Large-Scale Social Network Analysis

Mattia Lambertini Matteo Magnani Moreno Marzolla
Danilo Montesi Carmine Paolino

Technical Report UBLCS-2011-05

July 2011

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)

The University of Bologna Department of Computer Science Research Technical Reports are available in PDF and gzipped PostScript formats via anonymous FTP from the area `ftp.cs.unibo.it:/pub/TR/UBLCS` or via WWW at URL `http://www.cs.unibo.it/`. Plain-text abstracts organized by year are available in the directory `ABSTRACTS`.

Recent Titles from the UBLCS Technical Report Series

2009-12 *Natural deduction environment for Matita*, C. Sacerdoti Coen, E. Tassi, June 2009.

2009-13 *Hints in Unification*, Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E., June 2009.

2009-14 *A New Type for Tactics*, Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E., June 2009.

2009-15 *The k-Lattice: Decidability Boundaries for Qualitative Analysis in Biological Languages*, Delzanno, G., Di Giusto, C., Gabbrielli, M., Laneve, C., Zavattaro, G., June 2009.

2009-16 *Landau's "Grundlagen der Analysis" from Automath to lambda-delta*, Guidi, F., September 2009.

2010-01 *Fast overlapping of protein contact maps by alignment of eigenvectors*, Di Lena, P., Fariselli, P., Margara, L., Vassura, M., Casadio, R., January 2010.

2010-02 *Optimized Training of Support Vector Machines on the Cell Processor*, Marzolla, M., February 2010.

2010-03 *Modeling Self-Organizing, Faulty Peer-to-Peer Systems as Complex Networks* Ferretti, S., February 2010.

2010-04 *The qnetworks Toolbox: A Software Package for Queueing Networks Analysis*, Marzolla, M., February 2010.

2010-05 *QoS Analysis for Web Service Applications: a Survey of Performance-oriented Approaches from an Architectural Viewpoint*, Marzolla, M., Mirandola, R., February 2010.

2010-06 *The dark side of the board: advances in Kriegspiel Chess (Ph.D. Thesis)*, Favini, G.P., March 2010.

2010-07 *Higher-Order Concurrency: Expressiveness and Decidability Results (Ph.D. Thesis)*, Perez Parra, J.A., March 2010.

2010-08 *Machine learning methods for prediction of disulphide bonding states of cysteine residues in proteins (Ph.D. Thesis)*, Shukla, P., March 2010.

2010-09 *Pseudo-Boolean clustering*, Rossi, G., May 2010.

2010-10 *Expressiveness in biologically inspired languages (Ph.D. Thesis)*, Vitale, A., March 2010.

2010-11 *Performance-Aware Reconfiguration of Software Systems*, Marzolla, M., Mirandola, R., May 2010.

2010-12 *Dynamic Scalability for Next Generation Gaming Infrastructures*, Marzolla, M., Ferretti, S., D'Angelo, G., December 2010.

2011-01 *Server Consolidation in Clouds through Gossiping*, Marzolla, M., Babaoglu, O., Panzieri, F., January 2011 (Revised May 2011).

2011-02 *Adaptive Approaches for Data Dissemination in Unstructured Networks*, D'Angelo, G., Ferretti, S., Marzolla, M., January 2011.

2011-03 *Distributed Computing in the 21st Century: Some Aspects of Cloud Computing*, Panzieri, F., Babaoglu, O., Ghini, V., Ferretti, S., Marzolla, M., May 2011.

2011-04 *Dynamic Power Management for QoS-Aware Applications*, Marzolla, M., Mirandola, R., June 2011.

Large-Scale Social Network Analysis

Mattia Lambertini[1] **Matteo Magnani**[1] **Moreno Marzolla**[1]
Danilo Montesi[1] **Carmine Paolino**[1]

Technical Report UBLCS-2011-05

July 2011

Abstract

Social Network Analysis (SNA) is an established discipline for the study of groups of individuals with applications in several areas like economics, information science, organizational studies and psychology. In the last fifteen years the exponential growth of on-line Social Network Sites (SNSs) like Facebook, QQ and Twitter has provided a new challenging application context for SNA methods. However, with respect to traditional SNA application domains these systems are characterized by very large volumes of data, and this has recently led to the development of parallel network analysis algorithms and libraries. In this paper we provide an overview of the state of the art in the field of large scale social network analysis; in particular we focus on parallel algorithms and libraries for the computation of network centrality metrics.

1. Università di Bologna, Dipartimento di Scienze dell'Informazione, Mura A. Zamboni 7, I-40127 Bologna (Italy). Email: `lamberti@cs.unibo.it` (M. Lambertini); `matteo.magnani@cs.unibo.it` (M. Magnani); `marzolla@cs.unibo.it` (M. Marzolla); `montesi@cs.unibo.it` (D. Montesi); `cpaolino@cs.unibo.it` (C. Paolino)

1 Introduction

One of the reasons behind the enormous popularity gained by Social Network Sites (SNSs) like Facebook and Twitter can be found in the natural tendency of humankind to create social relationships, as already condensed many centuries ago by Aristotle in his famous quote "man is by nature a political animal". By analyzing the structure of these networks we can acquire knowledge that is impossible to gather by focusing only on the individuals or on the single relationships that compose them.

Since the publication of Moreno's book on sociometry [42], several studies have addressed the problem of analyzing complex social structures, giving birth to the interdisciplinary research field of Social Network Analysis (SNA). One of the most important results of SNA has been the definition of a set of measures that describe the role of single individuals with respect to their network of relationships. These so-called *centrality* measures are of great practical relevance since, e.g., they can be used to identify influential people with the potential of controlling the information flow inside communication networks. Since the pioneering work by Freeman [26], who in 1979 defined a set of geodesic centrality measures for undirected networks, the concept of centrality has proved its empirical validity several times during the years [53]. Further works, such as those by White et al. [53] and more recently by Opsahl et al. [44], extended Freeman's measures to more general network types, also with applications to recent on-line social media [51]. The main centrality metrics, i.e., degree, closeness, betweenness, and PageRank, are fundamental to increase our understanding of a network. Different metrics emphasize complementary aspects usually related to the propagation of information [39], but also to general applications like information searching [23] or recommendation systems. In fact, centrality measures are also used in the more general field of Complex Network Analysis for applications such as studying landscape connectivity to understand the movement of organisms, analyzing proteins and gene networks, studying the propagation of diseases, and planning urban streets for optimal efficiency.

The recent proliferation of on-line SNSs has been of major importance for SNA. SNSs such as blogs, multimedia hosting services and microblogging platforms are growing incredibly fast thanks to their ability to encourage the development of User Generated Content through networks of users; information perceived as worth sharing by their users, ranging from private life to political issues, is posted online and shared within a list of connections with the potential of creating discussions. From the terrorist attack in Mumbai in 2008 to the so-called Twitter revolution in Iran in 2009, on-line SNSs have proved to be a reliable and efficient solution to communicate and spread information, becoming an interesting case study of social phenomena. Being able to compute centrality metrics on these large networks is fundamental to understand the process of information diffusion and the role of their users.

The main problem with computing centrality metrics on on-line social networks is the typical size of the data. As an example, Facebook has now more than 750 million active users[1], Twitter has currently about 200 million users[2], and LinkedIn reached more than 100 million users[3]. If we consider networks of User Generated Content, Flickr has more than five billion photos[4] and in 2010 YouTube videos have been played more than 700 billion times[5], just to list a few statistics of this kind.

The large size of current social networks (as shown in the above examples) makes their quantitative analysis challenging. From the computational point of view, SNA represents social networks as graphs on which the metrics of interest are computed. Existing sequential graph algorithms can hardly be used due to space and time constraints. These limitations motivated an increasing interest in parallel graph algorithms for SNA [35]. Initial research focused on parallel algorithms for shared memory multi-processor systems [4, 36], because graph algorithms exhibit poor locality and thus run more efficiently on shared memory architectures. However, due to the high cost of shared memory multi-processor systems, recent research is focused on algorithms

1. http://www.facebook.com/press/info.php?statistics
2. http://www.bbc.co.uk/news/business-12889048
3. http://blog.linkedin.com/2011/03/22/linkedin-100-million/
4. http://blog.flickr.net/en/2010/09/19/5000000000/
5. http://www.youtube.com/t/press_statistics

for distributed memory systems [27, 31, 34, 30, 14].

This paper focuses on the description and analysis of parallel and distributed computation of centrality measures on large networks. As stated above, centrality measures play an important role in many research areas. This, combined with the increasing size of real-world social networks and the availability of multi-core commodity computing clusters, makes parallel graph algorithms increasingly important. This paper aims to (i) raise awareness on the existence of these powerful tools within the network analysis community; (ii) provide an overview of the most important and recent parallel graph algorithms for computation of centrality metrics; (iii) describe the main implementations of these algorithms; (iv) provide some experimental results concerning their scalability and efficiency. This work includes a general introduction on centrality measures and parallel graph algorithms, as well as up-to-date references, in order to act as a starting point for researchers seeking information on this topic.

The paper is organized as follows. The next two sections are brief introductions to SNA and parallel computing architectures. These sections cover the main topics needed to understand the rest of the paper and make it self-contained—readers already knowledgeable of these topics may skip them. Then we indicate the main parallel algorithms for the computation of centrality measures on large social graphs and review the main software libraries for the parallel computation of SNA centrality measures based on several criteria, so that it should be possible for professionals to match their problems with existing solutions. Finally, we present an experimental assessment of some parallel graph algorithms aimed at evaluating the scalability and efficiency of computationally demanding algorithms from the aforementioned libraries. Experimental results will help the readers to understand the potentials and limits of the current implementations, and in general will provide insights on the issues faced when dealing with graph algorithms on parallel systems. We conclude the paper with some final remarks.

2 Social Network Analysis

A social network is a set of people interconnected by social ties, e.g., friendship or family relationships. Although recent works on on-line SNA often use more complex models like multi-modal or multi-layer networks to represent multiple kinds of interconnected entities [20, 38], for the purpose of this paper we will represent a social network as a set of nodes connected by edges where nodes represent individuals and edges indicate their relationships.

Definition 1 (Social Network) *A* Social Network *is a graph (V, E) where V is a set of individuals and $E \subseteq V \times V$ is a set of social connections.*

Depending on the specific SNS we may use specific graph types: undirected, e.g., for Facebook where friendship connections are reciprocal, directed, e.g., for Twitter where *following* relationships are used, and weighted, e.g., for G+ where the strength of the connections is part of the data model. A complete treatment of these graph types lies outside the scope of this paper and is not necessary to introduce centrality measures, therefore in the following we will focus only on unweighted undirected graphs.

In this section we cover the two main aspects of social network modeling that are needed to understand the remaining of the paper. The first part of the section introduces the main centrality measures: degree, closeness, betweenness and page rank. Then we briefly introduce the main structural models describing the distribution of edges inside the network. This aspect is very important because the structure of the social network may influence the efficiency of the algorithms and real networks are not only very sparse but have also very specific internal structures. Knowledge of these structures is thus fundamental to test the behavior of network algorithms.

2.1 Centrality measures

In this section we introduce the main centrality measures. As we have aforementioned several variations of these metrics are possible depending on the specific kind of network. However, the objective is not to cover all possible variations but to understand the role of the main types of centrality. At the end of the section we provide an example with all these measures computed on a simple graph.

2.1.1 Degree Centrality

Degree centrality is one of the simplest centrality measures and is used to evaluate the local importance of a vertex within the graph. This measure is defined as the number of edges incident upon a node. Finding the individuals with high degree centrality is important because this is a measure of their popularity inside the network. Twitter celebrities may have hundreds of thousand followers, and in friendship-based networks like Facebook popular users may not correspond to public figures, making degree centrality very important and yet very simple to compute. Another relevant aspect regarding degree centrality is that in real networks there is a very small proportion of users with many connections, with the consequence that user filtering based on high degrees is very effective.

Definition 2 (Node Degree) *Given an undirected graph $G = (V, E)$, the degree $\delta(v)$ of a node $v \in V$ is the number of edges incident upon v.*

Definition 3 ((Normalized) Degree centrality) *Given a graph $G = (V, E)$, the Degree centrality $DC(v)$ of a node $v \in V$ is defined as:*

$$DC(v) \stackrel{def}{=} \frac{\delta(v)}{n-1}$$

2.1.2 Closeness Centrality

Closeness centrality provides an average measure of the proximity between a node and all other nodes in the graph. If we assume that the longest a path the lowest the probability that an information item (meme) will be able to traverse this path, it appears how the information produced by a node with high closeness centrality will have a higher probability to reach the other nodes of the network in a short time. Differently from degree centrality, the computation of closeness centrality requires a global view of the network.

Definition 4 (Closeness Centrality) *Given a connected graph $G = (V, E)$, let $d(u, v)$ be the distance (length of the shortest path) between $u, v \in V$. The Closeness centrality $CN(v)$ of a node $v \in V$ is defined as:*

$$CN(v) \stackrel{def}{=} \frac{n-1}{\sum_{u \in V \setminus v} d(u, v)}$$

2.1.3 Betweenness Centrality

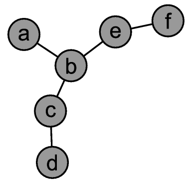
We have already mentioned that the information produced by popular users has a high probability of reaching many other users in the network. If these popular users have also a high value of closeness we may also expect that the process of information propagation will be fast. However, sometimes nodes that are not popular according to their degree centrality may nevertheless play an important role in propagating information: these users may act as bridges between separate sections of the network, having the potential to block the flow of information from one section to the other. These nodes are said to have a high value of betweenness centrality.

The definition of betweenness centrality is based on the concept of *pair-dependency* $\delta_{st}(v)$.

Definition 5 (Pair Dependency) *Given a connected graph $G = (V, E)$, let σ_{st} be the number of shortest paths between $s, t \in V$ and let $\sigma_{st}(v)$ be the number of shortest paths between s and t that pass through v. Pair-dependency $\delta_{st}(v)$ is defined as the fraction of shortest paths between s and t that pass through v. Formally:*

$$\delta_{st}(v) \stackrel{def}{=} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Betweenness centrality $BC(v)$ of a node v can be computed as the sum of the pair dependency over all pairs of vertices $s, t \in V$.



Node	PageRank	Degree	Closeness	Betweenness
a	0.105	0.2	0.45	0
b	0.282	0.6	0.71	8
c	0.198	0.4	0.56	4
d	0.109	0.2	0.38	0
e	0.198	0.4	0.56	4
f	0.109	0.2	0.38	0

Figure 1: An example of the main centrality measures

Definition 6 (Betweenness centrality) *Given a connected graph $G = (V, E)$, the betweenness centrality $BC(v)$ of a node $v \in V$ is defined as:*

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

2.1.4 PageRank

A limitation of degree centrality is the fact that it considers only the direct connections of a user. However there may be users with only a few connections that are very popular. This situation would not be captured using degree centrality. A possible extension to consider higher order relationships is PageRank centrality, corresponding to the famous algorithm proposed by Larry Page, Sergey Brin, Rajeev Motwani and Terry Winograd to rank Web pages [45]. As there is a very large body of literature focusing on this measure we do not provide additional details here.

2.2 Summary of the main centrality measures

In Figure 1 we have indicated an example of the main centrality measures. In this example the node d has the highest degree centrality because it is connected to the largest number of nodes (a, c and e). At the same time it is in the middle of the network, this corresponding to a high betweenness centrality, and it is on average closer to all the other nodes, determining its high closeness centrality.

2.3 Network models

Over the past fifty years several models have been proposed to explain the structure of social networks. Three models that are often used to test network algorithms are the *Random Network model*, the *Small-World model* and the *Scale-free model*.

The simplest model was proposed in 1959 by Erdos and Rényi, who introduced the concept of **Random network** [22]. Intuitively a random network is a graph $G = (N, E)$ in which pairs of nodes are connected with some fixed probability. A trivial algorithm to generate random graphs consists in looping through the nodes of the graph and create an edge (v, t) with some probability p for each pair of nodes $v, t \in E$.

Definition 7 (Erdös-Rényi Random network) *The Erdös-Rényi model $G_{n,p}$ is an undirected graph with n nodes where there exists an edge with probability p between any two distinct nodes.*

Although very interesting and studied from a formal point of view, random networks are not representative of many real social networks: empirical evidence shows that in real network the distance between any two nodes is lower on average than in random networks. This is often referred to as the phenomenon of the *six degrees of separation*. As a consequence another common model used for the generation of social network datasets is the **Small World model**. A small world network is a random network with the following properties:

- A short average path length.

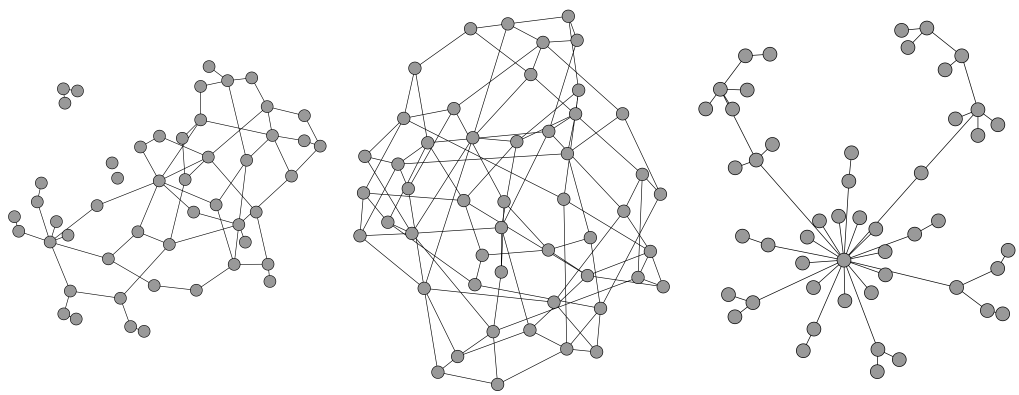


Figure 2: Three graphs with 50 nodes and different structures: from left to right, a random graph (with wiring probability: 0.05), a Watts-Strogatz small world network and a Barabàsi-Albert free-scale network

- A high clustering coefficient.

Many real world networks expose these properties and in 1998 Watts and Strogatz proposed an algorithm to produce this kind of network [50].

The Watts-Strogatz small world network model is the best known model to represent the Small World phenomenon. However in the last decade new important properties have been discovered concerning real networks. In 1999, Barabàsi and Albert [8] noticed that the degree distribution of many real world networks follows a power law and introduced the **Scale-free model**. In simple terms, they showed that in many real-world networks there are only a few nodes with a high degree and the number of nodes with a high degree decreases exponentially. The Barabàsi-Albert model is based on the mechanism known as **preferential attachment**[6].

In Figure 2 we have represented the three aforementioned kinds of networks.

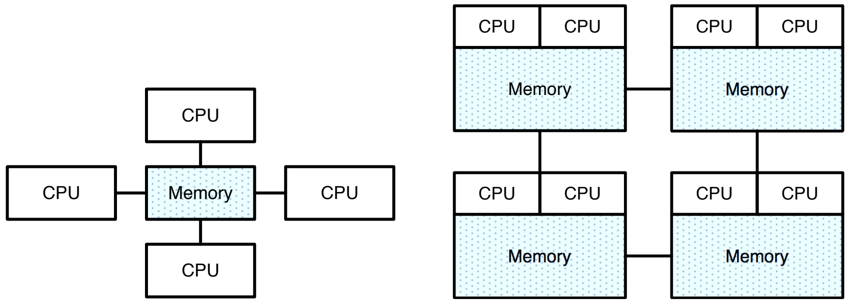
3 Parallel Computing

Thanks to the proliferation of Web 2.0 applications and SNSs, researchers have now the opportunity to study large, real social networks. The size of those networks, which often exceeds millions of nodes and billions of edges, requires tremendous amounts of memory and processing power to be stored and analyzed.

Typically, computer programs are written for serial computation: algorithms are broken into serial streams of instructions which are executed one after another by a single processor. Parallel computing, on the other hand, uses multiple processors simultaneously to solve a single problem. To do so, the algorithms are broken into discrete parts to be executed concurrently, distributing the workload among all the computing resources, therefore using less time.

Most parallel architectures are built from commodity components instead of custom, expensive ones. In fact, the six most powerful supercomputers of the world use mass-marketed processors[7]. This makes parallel computing an efficient, cost-effective solution to large-scale computing problems, promoting its widespread adoption among both the consumer and the scientific computing worlds.

6. Preferential attachment means that the probability that a new node A will be connected to an already existing node B is proportional to the number of edges that B already has.
7. `http://www.top500.org/lists/2010/11`



(a) UMA (b) NUMA

Figure 3: Schematic of shared memory architectures

Unfortunately, there are many classes of algorithms for which an efficient parallel implementation is not known. For other classes of problems, parallel algorithms do exist but do not scale well, meaning that the algorithms are unable to make efficient use of all the resources available. Graph algorithms, which are those used in network analysis, are an example of problems which are hard to parallelize efficiently, as we will see later.

In this section we give a general overview of parallel and distributed computing architectures and programming models, highlighting their advantages and disadvantages. We then focus our discussion on the challenges posed by parallel graph algorithms.

Many books cover parallel programming and architectures in detail [32, 17]; for a recent review of the parallel computing landscape see [3] and references therein.

3.1 Shared memory architectures

The last forty years have seen tremendous advances in microprocessor technology. Processor clock rates have increased from about 740 kHz (e.g., Intel 4004, circa 1971) to over 3 GHz (e.g., AMD Opteron, circa 2005). Keeping up with this exponential growth of computing power (as predicted by Moore's law [41]) has become extremely challenging as chip-making technologies are approaching fundamental physical limits [12].

Microprocessors needed a major architectural overhaul in order to improve performance. The industry found a solution to this problem in shared memory multi-processor architectures, which recently evolved into multi-core processors. In a shared memory multi-processor system there is a single, global memory space which can be accessed transparently by all processing elements. Communication and data transfer are implemented as read/write operations on the shared memory space. Shared memory multi-processors also provide suitable synchronization primitives (e.g., mutexes, memory barriers, atomic test-and-set instructions) to implement consistent updates.

Shared memory systems can be categorized by memory access times in Uniform Memory Access (UMA) and Non Uniform Memory Access (NUMA) machines.

Small multi-processor systems typically have a single connection between one of the processors and the shared memory, resulting in fast and uniform memory access across all processors. Such systems are called UMA machines. Figure 3a shows an example of a UMA machine with 4 processors.

In large shared memory multi-processor systems, those having tens or hundreds of computing nodes, each processor has local and non-local (local to another processor) memory. Therefore, memory access times depend on the memory which is accessed; these systems are in fact called NUMA machines. Figure 3b shows an example schematic of a NUMA machine with 8 processors and 4 memories.

Thanks to the global memory space, shared memory machines are easier to program than other parallel systems. Programmers do not have to worry about explicit data sharing and distribution among processors. Also, latencies are generally lower than in distributed memory systems (described below). However, shared memory machines are prone to hardware scalability problems—adding more processors leads to congestion of the internal connection network due to concurrent accesses to shared data structures and increasing memory write contention.

The most common way to program shared memory systems is by using threads. Threads are the smallest units of processing that can be scheduled by an operating system. In the thread programming model, each process can split into multiple concurrent executable programs. Each thread can be executed in a different processor, or in the same processor using time multiplexing, and typically shares memory with the other threads originated from the same process. Threads are commonly implemented with libraries of routines explicitly called by parallel code, such as implementations of the POSIX Threads standard, or compiler directives embedded in serial code, such as OpenMP[8]. OpenMP is a programming API for developing shared memory parallel code in Fortran, C, and C++ [43]; high level OpenMP directives can be embedded in the source code (e.g., signaling that all iterations of a "for" loop can be executed concurrently as there are no dependencies), and the compiler will automatically generate parallel code. Direct usage of threading libraries leads to greater flexibility, whereas semi-automatic generation of parallel code through OpenMP is often preferred in scientific computing thanks to its greater user-friendliness.

3.2 Massively multi-threaded computers

Massively multi-threaded computers are designed to tackle memory latency issues and context switch delays in a very different way than other parallel systems. Specifically, massively multi-threaded computers try to reduce the probability that processors become idle waiting for memory data by using a very large number of hardware threads, so that there is a chance that while some threads are blocked waiting for data, other threads can operate. Massively multi-threaded machines support very fast context switches (usually requiring a single clock cycle), dynamic load balancing, and word-level synchronization primitives. The latter is particularly important, since a large number of concurrent execution threads increases the probability of contention on shared data structures. Word-level synchronization allows the most fine control of contention.

An example of massively multi-threaded supercomputers are Cray's MTA-2 [1] and XMT [24] (also known as *Eldorado*). An MTA processor can hold the state of up to 128 instruction streams (threads) in hardware, and each stream can have up to 8 pending memory operations. Every processor executes instructions from a different non-blocked thread at each cycle, so it is fully occupied as long as there are sufficient active threads. Threads are not bound to any particular processor and the system transparently moves threads around to balance the load.

When an algorithm needs to know the global state, a synchronization must occur. On shared memory machines, synchronization is implemented in software and therefore it is an expensive operation. MTA machines solve this problem by implementing word-level synchronization primitives in hardware, drastically decreasing the cost of these operations, thus improving the system scalability.

Massively multi-threaded computers work best with algorithms that exhibit fine-grained parallelism, like most distributed graph algorithms [4]. Since the goal is to saturate the processors, the finer the level an algorithm can be parallelized, the more saturated the processors, the faster the program will run.

However, being made of custom processors on a custom architecture, MTA machines are extremely expensive and relatively slow. The latest Cray XMTs[9] includes 500 MHz Cray Threadstorm processors, which are several times slower than today's consumer multi-GHz processors. Having just 8 GB of memory per processor, Cray XMTs are also memory-constrained.

3.3 Distributed memory architectures

In contrast with the shared memory model, in the distributed memory model each processor has access to its own private memory only, and must explicitly communicate to other processors

8. `http://openmp.org`
9. `http://www.cray.com/Products/XMT/Product/Specifications.aspx`

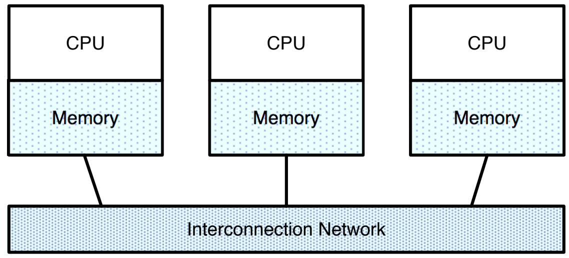


Figure 4: Schematic representation of the distributed memory architecture

when remote data is required. A distributed memory system consists of a network of homogeneous computers, as shown in Figure 4.

In a distributed memory system a process runs in a single processor, using local memory to store and retrieve data. Processes communicate using the message passing paradigm, which is typically implemented by libraries of communication routines. The *de facto* standard for message passing in parallel computing is Message Passing Interface (MPI) [40]. MPI is an Application Programmer Interface (API) and communication protocol standardized by the MPI Forum in 1994. It supports point-to-point and collective communication and it is language and platform-independent. Thus, many implementations of this interface exist and they are often optimized for the hardware upon which they run.

MPI can be combined with the thread model of computation (see Section 3.1) to exploit the characteristics of hybrid distributed/shared memory architectures, like clusters of multi-core computers.

The latest version of MPI (version 2.2) supports one-sided communication, collective extensions, dynamic process management, distributed I/O, along with many other improvements. However, only few implementations of MPI 2 exist.

Since the state of the running application is not shared across the system and every memory area is independent, distributed memory systems can be made less susceptible to interconnection overhead when scaling to hundreds, or even thousands of computing units. This is true as long as each individual node operates mostly on local data, and communication with remote nodes is not frequent.

One advantage of distributed memory architectures is that they can be built using commodity components like personal computers and consumer networks (e.g., Ethernet), whereas massive shared memory multi-processors are custom architectures. For this reason, distributed memory systems can be cost-effective when compared to other parallel architectures.

However, distributed memory systems are significantly harder to program than shared memory architectures because memory access times are non-uniform, differing several orders of magnitude from fast access to local memory, to slow network access to remote nodes. Therefore, programmers are responsible for distributing the data among all processors in a way that remote communication is reduced to a minimum. Unfortunately, as we will see shortly, this is not always possible, especially when dealing with problems that exhibit a fine-grained degree of parallelism. Graph algorithms are an example of these problems, in which most of the time is spent in fetching data from memory rather than executing CPU-bound computations.

3.4 Challenges in Parallel Graph Algorithms

Efficient parallel algorithms have been successfully developed for many scientific applications, with particular emphasis on solving large-scale numerical problems that arise in physics and engineering [49] (e.g., computational fluid dynamics, large N-body problems, computational chemistry and so on). As the size of real-world networks grow beyond the capability of a single processor, a natural solution is to start looking at parallel and distributed computing as a solution

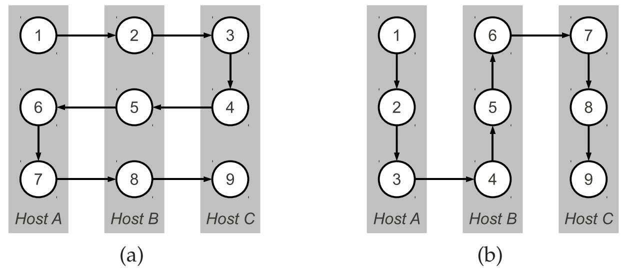


Figure 5: Different partitioning strategies may lead to very different performance of the same algorithm. Arrows denote (directed) graph edges.

to scalability problems of large scale network analysis.

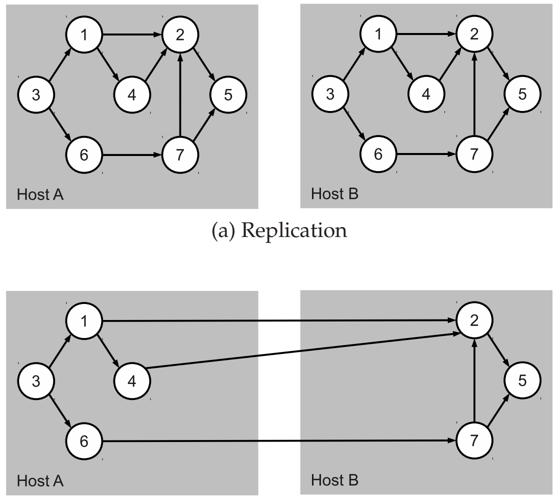
Unfortunately, the development of parallel graph algorithms—the "Swiss army knife" for large scale networks analysis—faces challenges which are related to the very nature of the problem. Specifically, in [35] the authors identify the following four main challenges towards efficient parallel graph processing: (i) data-driven computations; (ii) unstructured problems; (iii) poor locality; (iv) high data access to computation ratio.

Most graph algorithms are *data-driven*, meaning that the steps performed by the algorithm are defined by the structure of the input graph, and therefore can not be predicted. This requires parallel algorithms to discover parallelization opportunities at run-time, which is less efficient than hard-coding them within the code. For example, parallel dense vector-matrix multiplication involves the same data access pattern regardless of the specific content of the vector and matrix to be multiplied; on the other hand, parallel graph algorithms (e.g., minimum spanning tree computation) may behave differently according to the input data they need to operate on.

The issue above is complicated by the fact that input data for parallel graph algorithms is usually highly *unstructured*, which makes it harder to extract parallelism by partitioning the input data. Going back to the example of vector-matrix multiplication, the input data is easily partitioned by distributing equal sized blocks (or "strides") of the input matrix to the computing nodes, such that each processor can compute a portion of the result using local data only. Partitioning a graph across all processors is difficult, because the optimal partitioning depends both on the type of algorithm executed, and also on the structure of the graph.

Graph algorithms also exhibit poor *locality of reference*. As said above, the computation is driven by the node and edge structure of the graph. In general, most graph algorithms are based on visiting the nodes in some order, starting from one or more initial locations; the neighbors of already explored nodes have larger probability to be explored next. Unfortunately, it is generally impossible to guarantee that the neighbors of a node are laid out in memory such that locality of reference is preserved. The lack of locality is particularly problematic for distributed memory architectures, where non-local data accesses are orders of magnitude slower than local (RAM) memory access. As a simple example, let us consider the exploration of a simple graph with 9 nodes distributed across three different hosts. We assume that the graph forms a chain, such that there is a directed edge from node i to node $i + 1$. If we start the exploration from node 1, the partitioning shown in Figure 5a causes a server to pass control to other servers for 6 times, while the layout shown in Figure 5b causes servers to pass control 2 times.

Finally, graph algorithms tend to exhibit high *data access to computation ratio*. This means that a small fraction of the total execution time is spent doing computations, while most of the time is spent accessing data. Many algorithms are based on exploration of the graph. Rather than numerical computations as in other types of scientific workloads. Therefore, parallel graph algorithms tend to be communication bound, meaning that their execution time is dominated by the cost of communication among all processors. Communication bound algorithms are difficult to parallelize efficiently, especially in parallel systems with high communication costs.



(a) Replication

(b) Partitioning

Figure 6: Graph replication and partitioning example

3.5 Addressing locality issues in distributed memory architectures

As observed above, lack of locality and high data access to computation ratio make efficient implementation of parallel graph algorithms problematic. This problem is particularly severe for distributed memory architectures, which unfortunately are also the most appealing systems for parallel computing due to their low cost. Therefore, it is useful to spend a few words on distributed data layout for graphs.

First, we consider graphs represented using adjacency lists, which means that the graph is encoded as a set of nodes, each node having an associated set of incident edges. Alternative representations are possible, for example using adjacency matrices, where a n node graph is encoded as a $n \times n$ (usually sparse) matrix whose nonzero elements denote edges.

One possibility is to store a copy of the entire graph on each processing node (Figure 6a). This has the advantage that each processor does not need to interact or pass control to other processors to explore the graph; however, synchronization with other processors might still be required by the specific algorithm being executed. Replication is particularly effective for task-parallel algorithms, that are those algorithms which can be decomposed into mostly independent tasks. For example, the computation of betweenness centrality on an n node graph using Brande's algorithm [13] requires n independent visits, each one starting from a different node. This algorithm can be efficiently implemented on distributed memory architectures by replicating the input graph and assigning to each processor the task of visiting the graph from a subset of the nodes.

Replication has the serious disadvantage of not making optimal use of the available memory: in fact, each computing node must have enough RAM to keep a copy of the whole graph. This is not always possible, since large real-world graphs are larger than the RAM available on any conceivable single computing node. To address this issue, it is possible to partition the input graph, such that each processing node is responsible for storing only part of the graph (Figure 6b). This results in the useful side effect that it is possible to handle larger graphs by simply adding more computing nodes. The downside is that communication costs might limit scalability, if the partitioning is not done accurately. Some performance results will be discussed in Section 6.

Algorithm 1 Sequential Floyd-Warshall

Input: $M_i j$ adjacency matrix for graph $G = (V, E)$
  for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to n **do**
      $d(i, j) := M_{ij}$
  for $k \leftarrow 1$ to n **do**
    for $i \leftarrow 1$ to n **do**
      for $j \leftarrow 1$ to n **do**
        $d(i, j) \leftarrow \min(d(i, j), d(i, k) + d(k, j))$

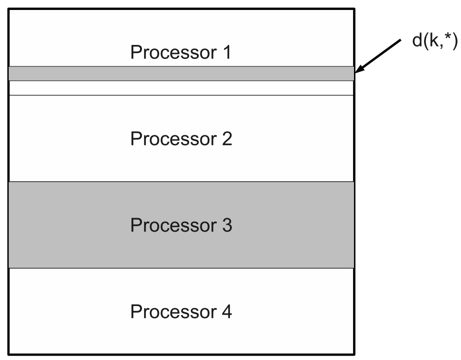


Figure 7: Computing all-pair shortest paths on four processors; shaded areas denote the portions of matrix $d(i, j)$ which must be stored within processor 3

4 Parallel computation of centrality measures

In this section we briefly describe some parallel algorithms for computation of centrality measures in social graphs. Among the metrics mentioned in Section 2, betweenness centrality is the most challenging to compute efficiently using parallel algorithms [5, 21].

Parallel computation of degree and closeness centrality is relatively easy. In fact, computation of the degree centrality is almost an *embarrassingly parallel* task, which means that the computation can be split across p processors such that each processor can compute part of the result independently and without the need to interact with other processors.

The degree centrality $DC(v)$ of a node v is simply the node degree $\delta(v)$ divided by $n - 1$, n being the number of nodes in the graph. Therefore, given a graph $G = (V, E)$, it is possible to partition V into n/p disjoint subsets, and assign each subset to one of the p processors. At this point, each processor trivially computes the degree of every node in its partition. This algorithm does not require any communication between processors, so it can be efficiently implemented even on distributed memory architectures with partitioned input graph. For example, host A in Figure 6b can compute the degree centrality of nodes $\{1, 3, 4, 6\}$, while host B can compute the degree centrality of nodes $\{2, 5, 7\}$.

Closeness centrality $CN(v)$ involves the computation of all pairwise distances $d(u, v)$ for all nodes $u, v \in V$. Most of the existing parallel all-pair shortest path algorithms are based on either the Floyd-Warshall algorithm, or on Dijkstra's Single Source Shortest Path (SSSP) algorithm which is executed n times, one for each node $u \in V$. In both cases, efficient parallel implementations are well known [25, Chapter 3.9].

We briefly describe a parallel version of the Floyd-Warshall algorithm; the sequential version is very simple, and is described by Algorithm 1. M_{ij} is the adjacency matrix for graph G, such that M_{ij} is the length of edge (i, j), if such an edge exists, or $+\infty$ otherwise. The sequential algorithm performs a set of relaxation steps, updating the distance estimates $d(u, v)$.

A simple approach to parallelize Algorithm 1 is to partition the matrix d row wise, and assign each block of n/p contiguous rows to each of the p processors. In addition to this local data, each processor needs a copy of the k-th row $d(*, k)$ to perform its computations; at each iteration, the processor that has this row can broadcast it to all other processors. Figure 7 shows an example of data partitioning across four processors.

We now turn our attention to the computation of betweenness centrality. At the time of writing the best known sequential algorithm is due to Brandes [13]. If we define $\delta_s(v) = \sum_{s \neq v \neq t \in V} \delta_{st}(v)$, then the betweenness score of node $v \in V$ can be then expressed as

$$BC(v) = \sum_{s \neq v \in V} \delta_s(v)$$

Let $P_s(v)$ denote the set of predecessors of a vertex v on shortest paths from s, defined as:

$$P_s(v) = \{u \in V : (u, v) \in E, d(s, v) = d(s, u) + w(u, v)\}$$

Brandes shows that the dependencies satisfy the following recursive relation, which is the key idea of the algorithm:

$$\delta_{s*}(v) = \sum_{w, v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_{s*}(w))$$

Based on the facts above, $BC(v)$ can be computed in two steps: first, execute n SSSPs, one for each source $s \in V$, maintaining the predecessor sets $P_s(v)$; then, compute the dependencies $\delta_{s*}(v)$ for all $v \in V$. At the end we can compute the sum of all dependency values to obtain the centrality measure of v.

In 2006, Bader and Madduri [5] proposed a parallel algorithm for betweenness centrality, which exploits parallelism at two levels: the SSSP computation from each source vertex is done concurrently, and each individual SSSP is also parallelized (see Algorithm 2).

The algorithm assigns a fraction of the graph nodes to each processor, which can then initiate the SSSP computation. The stack S, list of predecessors P and the Breadth-First Search (BFS) queue Q are replicated on each computing node, so that every processor can compute its own partial sum of the centrality value for each vertex, and all the sums can be merged in the end using a reduction operation. Algorithm 2 is not space efficient, as it requires storing the whole SSSP tree for each source node.

The algorithm above requires fine-grained parallelism for update the shared data structures, and is therefore unsuitable for a distributed memory implementation. Edmonds, Hoefler, and Lumsdaine [21] recently proposed a new parallel space-efficient algorithm for betweenness centrality which partially addresses these issues: the new algorithm requires coarse-grained parallelism, and therefore is better suited for distributed memory architectures. Furthermore, memory requirements are somewhat reduced.

5 Libraries

In this section we describe some of the available software libraries and tools for graph analysis on parallel and distributed architectures—sequential network analysis packages are not considered here. The list reported here is not meant to be exhaustive. However, we tried to cover some of the most relevant parallel graph analysis packages available, both for shared memory and for distributed memory architectures. All software packages described here are distributed under free software licenses (generally, GNU General Public License or BSD-like licenses).

Parallel Boost Graph Library (PBGL) The PBGL [27] is a library for distributed graph computation which is part of Boost [11], a collection of open source, peer-reviewed libraries written in C++. The PBGL is based on another Boost library, the serial Boost Graph Library (BGL) [48], offering similar syntax, data structures, and algorithms. Like all the Boost libraries, it is distributed under the Boost Software License, a BSD-like permissive free software license.

Algorithm 2 Bader and Madduri parallel betweenness centrality

Input: $G(V, E)$
Output: Array $BC[1..n]$, where $BC[v]$ is the centrality value for vertex v

```
for all v ∈ V in parallel do
    BC[v] ← 0;
for all s ∈ V in parallel do
    S ← empty stack;
    P[w] ← empty list, ∀w ∈ V;
    σ[t] ← 0, ∀t ∈ V;
    σ[s] ← 1;
    d[t] ← −1, ∀t ∈ V;
    d[s] ← 0;
    Q ← empty queue;
    enqueue s to Q;
    while Q not empty do
        dequeue v from Q;
        push v to S;
        for each neighbor w of v in parallel do
            if d[w] < 0 then
                enqueue w → Q;
                d[w] ← d[v] + 1;
            if d[w] = d[v] + 1 then
                σ[w] ← σ[w] + σ[v];
                append v → P[w];
    δ[v] ← 0, ∀v ∈ V ;
    while S not empty do
        pop w ← S;
        for v ∈ P[w] do
            δ[v] ← δ[v] + (σ[v]/σ[w])(1 + δ[w])
        if w ≠ s then
            BC[w] ← BC[w] + δ[w]
```

The PBGL aims at being efficient and flexible at the same time by employing the generic programming paradigm. Its generic algorithms are defined in terms of collections of properties called *concepts*. A concept defines a set of type requirements extracted from an algorithm, in order to ensure the computational feasibility and efficiency of that algorithm on user-provided types. An example is the *Graph* concept, which express the fact that a graph is a set of vertices and edges with some additional identification information. This makes the PBGL flexible enough to work with parametric data structures; since those abstractions are removed by the compiler, a generic algorithm can be as fast as an hard-coded one.

In order to decouple vertex and edge properties accessors and their representation, the PBGL implements *property maps*. Thanks to property maps, vertex and edge properties can be stored either within the graph data structure itself, or as distinct, arbitrary data structures. Property maps can be distributed in such a way that each processor only maintains the values for a subset of nodes.

The PBGL implements some important measures for social network analysis, like betweenness centrality, as well as several fundamental graph algorithms, including BFS, Depth-First Search (DFS), SSSP, Minimum Spanning Tree (MST), Connected Components (CC), and *s-t* connectivity which determines whether there is a path from vertex s to vertex t on a given graph.

Small-world Network Analysis and Partitioning (SNAP) SNAP [37, 6] is a parallel library for graph analysis and partitioning, targeting multi-core and massively multi-threaded platforms. SNAP is implemented in C, using POSIX threads and OpenMP primitives for parallelization; the software is distributed under the GPLv2 license.

SNAP pre-processes input data in order to choose the best available data structure: the default are cache-friendly adjacency arrays, but switches to dynamically resizable adjacency arrays when dynamic structural updates are required, and sorts them by vertex or edge identifier when fast deletions are necessary.

SNAP is specifically optimized for processing small-world networks by exploiting their characteristics, such as low graph diameter, sparse connectivity, and skewed degree distribution. For example, SNAP represents low-degree vertex adjacencies in unsorted adjacency arrays, and high degree vertices in a structure similar to a randomized binary search tree called treap [2], for which efficient parallel algorithms for set operations exist.

On the algorithm side, SNAP fundamental graph algorithms such as BFS, DFS, MST, and CC are designed to exploit the fine-grained thread-level parallelism offered by shared memory architectures. SNAP also supports other SNA metrics that have a linear or sub-linear computational complexity such as average vertex degree, clustering coefficient, average shortest path length, rich-club coefficient, and assortativity. These metrics are also useful as preprocessing steps for optimizing some analysis algorithms.

Multi-Threaded Graph Library (MTGL) The MTGL [47, 10, 9] is a parallel graph library designed for massively multi-threaded computers. It is written in C++ and distributed under the MTGL License, a BSD-like custom license. It is inspired by the generic programming approach of the BGL and PBGL, but rather than maximizing its flexibility, it is designed to give developers full access to the performance of MTA machines. Due to the high cost and low availability of massively multi-threaded computers, the MTGL has been extended [9] to support Qthreads [52], a library that emulates the MTA architecture on standard shared memory computers. While scalability and performance degrade when using standard shared memory architectures, Barrett et al. [9] have shown that performance issues and benefits are similar in both cases. The MTGL provides a wide range of graph algorithms, including BFS, DFS, CC, Strongly Connected Components (SCC), betweenness centrality, community detection and many others.

HIPG HIPG [29, 31] is a distributed framework for processing large-scale graphs. It is implemented in Java using the Ibis message-passing communication library [7], and it is distributed under the GPLv3 license. HIPG provides an high-level object-oriented interface that does not expose explicit communication. Nodes are represented by objects and can contain custom fields to represent attributes and custom methods. However, due to Java objects memory overhead, edges are stored in a single large integer array, distributed in chunks across all computing nodes; access to this data structure is managed by an abstraction layer in order to be transparent to the user. An HIPG distributed computation is defined by plain Java methods executed on graph nodes; each call to a method of a remote node (i.e., a node not stored locally on the caller processor) is automatically translated in an asynchronous message. In order to compute aggregate results, HIPG uses logical objects called *synchronizers* that can apply reduction operations. Synchronizers can also manage the computation by calling methods on nodes and setting up barriers. Synchronizers can be hierarchically organized to support sub-computations on sub-graphs, supporting divide-and-conquer graph algorithms.

DisNet DisNet [34, 19] is a distributed framework for graph analysis written in C++ and distributed under the GPLv3 license. Instead of relying on MPI, DisNet implements its own message passing system using standard sockets, but provides an API that abstracts away all details of parallelism. DisNet is a master-worker framework: the master coordinates the workers and combine results, and workers run a user-specified routine on vertices. Workers communicate only with the master and never between each other. DisNet does not partition the graph, so every worker has a copy of the entire network. While this ensures an high level of efficiency, especially on data structure that exhibits poor locality (like graphs, see Section 3.4), it can be a problem with networks starting with millions of nodes and billions of edges. For example, a real social network with 4,773,246 vertices and 29,393,714 edges required 10 GB of memory [34]. Aside from those potential problems, DisNet is a viable alternative to the other libraries and tools we discussed in this chapter.

PeGaSus PeGaSus [30, 46] is an open source graph mining system implemented on the top of Hadoop [28], an open source implementation of the MapReduce framework [18]. PeGaSus provides algorithms for typical graph mining tasks, such as computation of diameter, CC, PageRank and so on. The algorithms above are implemented around matrix-vector multiplications; PeGaSus provides an efficient primitive for that, called Generalized Iterated Matrix-Vector multiplication (GIM-V). The GIM-V operation on the matrix representation of a graph with n nodes and m edges requires time $O\left(\frac{n+m}{p}\log\frac{n+m}{p}\right)$ using p processors.

Combinatorial BLAS The Combinatorial BLAS [14, 16] is a high performance distributed library for graph analysis and data mining. Like the Basic Linear Algebra Subroutines (BLAS) library [33], it defines a simple set of primitives which can be used to implement complex computations. However, the Combinatorial BLAS focuses on linear algebra primitives targeted at graph and data mining applications on distributed memory clusters. The Combinatorial BLAS implements efficient data structures and algorithms for processing distributed sparse and dense matrices, acting as a foundation to other data structures such as the sparse adjacency matrix representation of graphs. The library provides a common interface and allows users to implement a new sparse matrix storage format, without any modification to existing applications. It uses the MPI library to handle communication between computing nodes. Currently, the Combinatorial BLAS also includes an implementation of the betweenness centrality algorithm for directed, unweighted graphs, and a graph clustering algorithm.

We summarize in Table 1 the main features of the software packages above.

6 Performance considerations

We now present some performance results for two parallel implementations of the betweenness centrality algorithm: one for shared memory architectures and another for distributed memory architectures. We remark that the aim of this section is not to do an accurate performance analysis of the algorithms considered; instead, we want to show how the issues described in Section 3.4 can influence the scalability of parallel graph algorithms. We focus on betweenness centrality because it is a widely used metric, it is computationally demanding, and because implementations for distributed memory and shared memory architectures are readily available.

We consider two parallel implementations of the betweenness centrality algorithm. The first one is part of the PBGL [27] (included in Boost version 1.46.0) and is run on a distributed memory commodity cluster made of Intel PCs connected through a 100Mbps fast Ethernet LAN, using MPI as the communication framework. The second implementation is part of SNAP [6] version 0.4, and is run on a single node of an IBM pSeries 575 supercomputer hosted at the CINECA supercomputing center [10]. While the IBM 575 is actually a *distributed memory* system (our installation has 158 computing nodes), a single node has 32 cores with 128 GB of shared memory, hence it can be viewed as a 32-way shared memory system. Table 2 describes the details of the machines used in the tests.

We study the *relative speedup* S_p of both implementations as a function of the number p of processing cores. The relative speedup is a measure of the scalability of a parallel algorithm, and is defined as follows. Let T_p be the execution time of a parallel program run on p processors; the relative speedup S_p is the ratio of the execution time on 1 processor and the execution time on p processors:

$$S_p = \frac{T_1}{T_p}$$

If the total work can be evenly partitioned across all processors and there is no communication overhead, the execution time with p processor is approximately $1/p$ times the execution time with one processor. In such (ideal) conditions, a parallel algorithm exhibits perfect (or linear) speedup $S_p \approx p$. Unfortunately, in practice we have $S_p \ll p$, due to the following reasons:

10. http://www.cineca.it/

Table 1: Summary of Parallel Graph Libraries

Name	Latest Release	License	Lang.	Parallel API	Implemented Algorithms	Input Formats	Output Formats
PBGL [11, 27]	1.47.0 (Jul 2011)	BSD-like	C++	Distributed Memory (MPI)	BFS, DFS, Dijkstra's SSSP (+ variants), MST (+ variants), CC, SCC, PageRank, Boman et al. graph coloring, Fruchterman Reingold force-directed layout, *s*-*t* connectivity, Betweenness Centrality	METIS, API	GraphViz, API
SNAP [37, 6]	0.4 (Aug 2010)	GPLv2	C	Shared Memory (OpenMP)	BFS, DFS, CC, SCC, Diameter, Vertex Cover, Clustering Coefficient, Community Kullback Liebler, Community Modularity, Conductance, Betweenness Centrality, Modularity Betweenness, Modularity Greedy Agglomerative, Modularity Spectral, Seed Community Detection	DIMACS, GML, GraphML, METIS, SNAP, API	API
MTGL [47, 9]	1.0 (Jun 2011)	BSD-like	C++	Shared Memory (MTA architecture or Qthreads library)	BFS, Psearch (DFS variant), SSSP, CC, SCC, PageRank, Subgraph isomorphism, Random walk, *s*-*t* connectivity, Betweenness Centrality, Community Detection, Connection Subgraphs, Find triangles, Assortativity, Modularity, MaxFlow	Binary, DIMACS, Matrix Market, Memory Map, API	Binary, DIMACS, Matrix Market, Memory Map, API
DisNet [34, 19]	N/A (Jun 2010)	GPLv3	C++	Distributed Memory (custom)	Degree Centrality, Betweenness Centrality, Closeness Centrality, Eccentricity	DisNet, Adjacency List, Pajeck (variant), API	API
HipG [31, 29]	1.5 (Apr 2011)	GPLv3	Java	Distributed Memory (Ibis)	BFS, SCC, PageRank	SVC-II, Hip, API	SVC-II, Hip, API
PeGaSus [30, 46]	2.0 (Sep 2010)	BSD-like	Java	Distributed Memory (Hadoop)	Degree Centrality, PageRank, Random walk with restart, Radius, CC	Tab-separated	Plot
CombBLAS [14, 16]	1.1 (May 2011)	BSD-like	C++	Distributed Memory (MPI-2)	Matrix Operations, Betweenness Centrality, MCL graph clustering	N/A	N/A

	Distributed Memory	Shared Memory
Model	Commodity Cluster	IBM pSeries 575
Nodes	61	168
CPU type	Intel Core2 Duo E7 2.93 GHz	IBM Power6 4.7 GHz
Cores per node	2	32
RAM per node	2 GB	128 GB
Network	100Mb Fast Ethernet (half duplex)	Infiniband 4x DDR
Operating System	Linux 2.6.28-19	AIX 6
C/C++ Compiler	GCC 4.3.3	IBM XL C/C++ 11.1.0.8
Software Library	Boost 1.46.0	SNAP 0.4

Table 2: Technical specifications of the machines used for the tests

- Partitioning the input data across all processors is usually a task which must be done serially; in some cases, this task can become the bottleneck.
- If the input data is unstructured and irregular (graph data fall in this category), it may be difficult to partition it such that the workload is evenly distributed across all processors. In such cases, the slower processor may block the faster ones, introducing unwanted latencies and reducing scalability.
- Communication and synchronization costs can become a major issue, especially with a large number of processors. At some point a parallel program may experience *negative scalability*, meaning that the execution time grows as more processors are added.

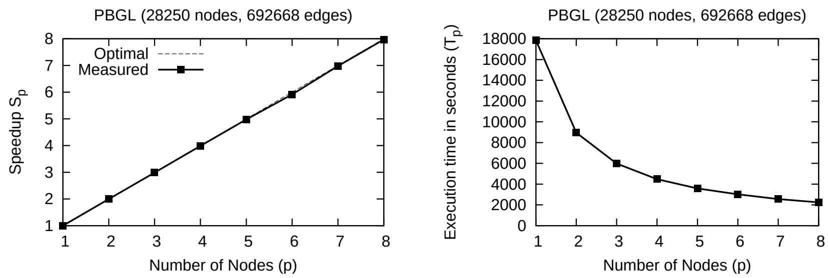
We now illustrate and discuss the results we have obtained on our test infrastructures.

Betweenness centrality on distributed memory architectures We first consider the betweenness centrality algorithm on a distributed memory cluster. We use the implementation of the centrality algorithm provided by the PBGL (included in Boost 1.46.0), which uses MPI as the communication layer. The program is executed on a commodity cluster made of Intel PCs running Debian Linux, connected using Fast Ethernet. This low-end solution is poorly suited for communication-intensive high performance computing tasks; however, similar infrastructures are readily available at most institutions, so it is important to test whether they can be used for large scale network analysis.

In the following tests we used a publicly available real social network dataset with 28250 nodes and 692668 edges structured as a scale-free graph [15]. The size of the graph is motivated by the need to get reasonable computation times: computing the centrality values of all vertices on a single CPU core requires about 5 hours on our infrastructure. The input graph was fully replicated on each node.

We also used an increasing number p of computing nodes, from $p = 1$ to $p = 8$. Larger values of p do not yield any significant advantage on the graph above. We run a single process on each machine (although all processors are dual core), since we do not want to mix local (inter-node) communications with remote (intra-node) ones. The execution times T_p with p processing nodes have been computed by averaging the results obtained over multiple independent experiments. Of course, all tests have been executed when all nodes were idle.

Figure 8 shows the speedup S_p and average execution times T_p with p processors. The algorithm achieves optimal (linear) speedup with the replicated input graph. This result is quite remarkable, considering that the test cluster has a poor network connection. The reason is that the parallel centrality algorithm has a low communication to computation ratio: each processor is assigned the computation of an SSSP, which can be computed using local data only. Inter-node communications happen only at the end of each SSSP, when each node updates the common



(a) Speedup (b) Execution Time

Figure 8: Speedup and Execution Time of the betweenness centrality algorithm in Boost, executed on the Intel cluster. The input graph (28250 nodes, 692668 edges) is replicated across the computing nodes.

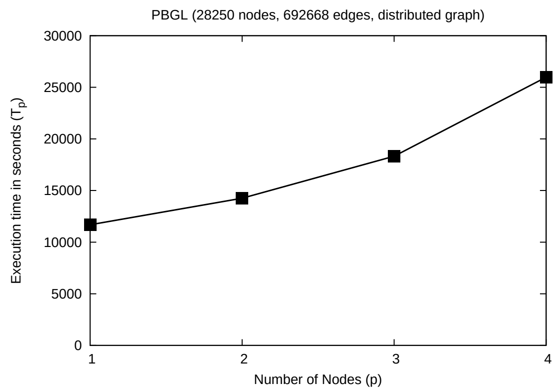


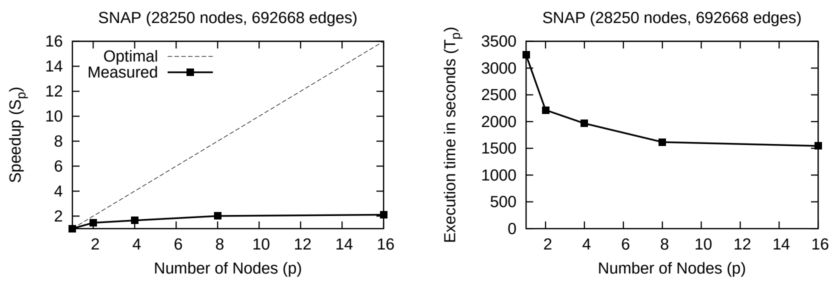
Figure 9: Execution time for a Small-World graph with 28250 nodes and 692668 edges. The input graph is distributed across the computing nodes.

data structure of centrality estimates. At the end of the algorithm, this data structure contains the exact centrality values for each node.

To achieve good scalability on the cluster, it is essential to reduce communications as much as possible. If the input graph is distributed across all processors, the situation changes dramatically. Figure 9 shows the execution time of the betweenness centrality algorithm on the same cluster as above, using a distributed storage model for the input graph. This means that each host only stores a subset of the graph; therefore, each SSSP computation must pass control to different processors, as the graph node being visited at each step may reside on a remote host. In this situation we observe negative scalability, since the execution time grows as more processors are added. Code profiling confirms that the algorithm is communication bound, which means that a significant fraction (about 90% in our case) of the execution time is spent waiting for data to be sent or received through the slow network.

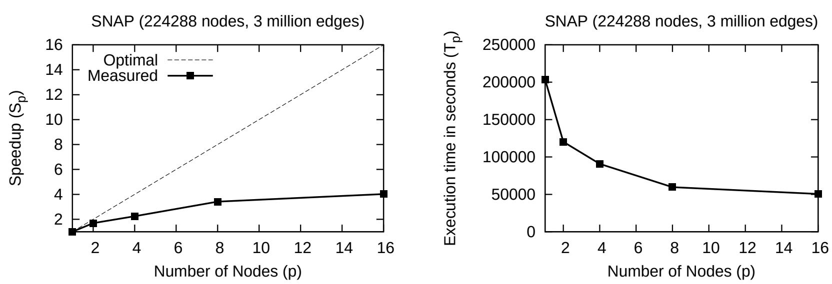
Betweenness centrality on shared memory architectures We now study the scalability of a different implementation of the betweenness centrality algorithm. Specifically, we consider the implementation provided by SNAP version 0.4 for shared memory architectures. The algorithm has been applied to the same graph (28250 nodes and 692668 edges), and has been executed on a single node of the IBM pSeries 575 supercomputer. The node has 32 processor cores sharing a common memory, therefore it can be seen as a shared memory system.

Figure 10 shows the speedup and total execution time with p processor cores, $p \in \{1, 2, 4, 8, 16\}$.



(a) Speedup (b) Execution Time

Figure 10: Speedup and Execution Time of the betweenness centrality algorithm in SNAP on the IBM p575, for a Small-World graph with 28250 nodes and 692668 edges.



(a) Speedup (b) Execution Time

Figure 11: Speedup and Execution Time of the betweenness centrality algorithm in SNAP on the IBM p575, for a larger Small-World graph with 224288 nodes and 3 million edges.

The scalability is very limited: with $p = 16$ cores, the algorithm requires about 47% the time required with a single core. In Figure 11 we show the performance results of the betweenness centrality algorithm of SNAP on a larger graph with 224288 nodes and 3 million edges.

While communication on a shared memory multi-processor is much more efficient than in commodity distributed memory clusters, memory access is still a bottleneck for data-intensive applications, since the memory bandwidth can quickly become inadequate to feed all processors.

Shared memory vs distributed memory. We conclude this section by reporting the results of a direct comparison between the PBGL and SNAP, both running on the commodity cluster, using the graph with 28250 nodes and 692668 edges. The distributed memory version is executed on p processors, from $p = 1$ to $p = 8$, using full replication of the input graph on all hosts. SNAP has been executed on a single host of the cluster using both processor cores, requiring about $1400s$ to process the whole graph.

Figure 12 shows the execution times of the two programs. It is interesting to observe that SNAP, using only two cores of a single CPU, is faster than PBGL running on $p = 8$ nodes for this graph. Scalability is an important metric, but should not be considered alone: a less scalable algorithm may be faster in practice than a perfectly scalable one, as this test demonstrates.

7 Concluding remarks

In this paper we considered the problem of computing centrality measures in large social networks using parallel and distributed algorithms. We first introduced the main centrality measures used in SNA. Then, we gave an overview of the main features and limitations of current

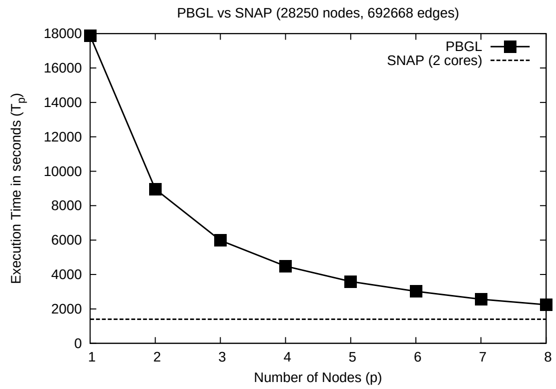


Figure 12: Execution time of the betweenness centrality algorithm provided by the PBGL and by SNAP. Both algorithms have been run on the commodity cluster; SNAP has been executed on a single server using both CPU cores.

parallel and distributed architectures, including distributed memory, shared memory and massively multi-threaded machines. We then briefly described some of the existing parallel algorithms for computing useful centrality measures. After that, we presented a set of software packages that implement such algorithms. Finally, we gave some insights on the performance of the betweenness centrality algorithms provided by the PBGL (on distributed memory architectures), and by SNAP (on shared memory architectures).

Parallel graph algorithms are a hot research topic, which is still waiting to make a real breakthrough. Graph algorithms exhibit a number of properties which make them very hard to parallelize efficiently on current high performance computing architectures: the large size of social graphs makes it very difficult to store them in RAM, and distributed storage of graph data across multiple computing nodes raises other issues, related to the long access times to remote graph data. Graph algorithms also exhibit a large communication to computation ratio, making them poor candidates for parallel implementation.

Despite the issues above, more efficient parallel architectures, combined with state-of-the-art algorithms, may open the possibility of managing large social network graphs, enabling scientists to get a better understanding of the social phenomena which, directly or indirectly, influence our lives.

References

[1] W. Anderson, P. Briggs, C. S. Hellberg, D. W. Hess, A. Khokhlov, M. Lanzagorta, and R. Rosenberg. Early experience with scientific programs on the cray mta-2. In *Proc. 2003 ACM/IEEE conference on Supercomputing*, SC'03, pages 46–, New York, NY, USA, 2003. ACM.

[2] C. R. Aragon and R. G.Seidel. Randomized search trees. In *Foundations of Computer Science, Annual IEEE Symposium on*, pages 540–545, Los Alamitos, CA, USA, 1989. IEEE Computer Society.

[3] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52:56–67, October 2009.

[4] D. A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *Proc. Int. Conf. on Parallel Processing*, pages 523–530, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[5] D. A. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *Proc. 2006 International Conference on Parallel Processing*, ICPP '06, pages 539–550, Washington, DC, USA, 2006. IEEE Computer Society.

[6] D. A. Bader and K. Madduri. SNAP, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks. In *Proc. Int. Symposium on Parallel and Distributed Processing*, IPDPS, pages 1–12, Miami, FL, Apr.14–18 2008.

[7] H. E. Bal, J. Maassen, R. V. van Nieuwpoort, N. Drost, R. Kemp, N. Palmer, G. Wrzesinska, T. Kielmann, F. Seinstra, and C. Jacobs. Real-world distributed computing with Ibis. *Computer*, 43:54–62, 2010.

[8] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):11, 1999.

[9] B. W. Barrett, J. W. Berry, R. C. Murphy, and K. B. Wheeler. Implementing a portable Multi-threaded Graph Library: The MTGL on Qthreads. In *IEEE International Symposium on Parallel & Distributed Processing*, IPDPS, pages 1–8, Rome, Italy, May23–29 2009.

[10] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Graph software development and performance on the MTA-2 and Eldorado. In *48th Cray Users Group Meeting*, Switzerland, Jan. 2006.

[11] Boost C++ Libraries, July 2011. Available at `http://www.boost.org/`.

[12] S. Borkar. Design challenges of technology scaling. *Micro, IEEE*, 19(4):23 –29, 1999.

[13] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[14] A. Buluç and J. R. Gilbert. The combinatorial BLAS: design, implementation, and applications. *International Journal of High Performance Computing Applications*, May19 2011.

[15] F. Celli, F. Di Lascio, M. Magnani, B. Pacelli, and L. Rossi. Social network data and practices: The case of friendfeed. In S.-K. Chai, J. Salerno, and P. Mabry, editors, *Advances in Social Computing*, volume 6007 of *LNCS*, pages 346–353. Springer Berlin/Heidelberg, 2010.

[16] Combinatorial BLAS Library (MPI reference implementation), May 2011. Version 1.1, Available at `http://gauss.cs.ucsb.edu/~aydin/CombBLAS/html/index.html`.

[17] D. Culler, K. P. Singh, and A. Gupta. *Parallel Computer Architecture–A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[18] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[19] DisNet, A Framework for Distributed Graph Computation, 2011. Available at `http://nd.edu/~dial/software.html`.

[20] N. Du, H. Wang, and C. Faloutsos. Analysis of large multi-modal social networks: patterns and a generator. In *Proceedings of the 2010 European conference on Machine learning and knowledge discovery in databases: Part I*, ECML PKDD'10, pages 393–408, Berlin, Heidelberg, 2010. Springer-Verlag.

[21] N. Edmonds, T. Hoefler, and A. Lumsdaine. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *Proc. Int. Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, Dec. 2010.

[22] P. Erdős and A. Rényi. On random graphs I. *Publ. Math. Debrecen*, 6(290-297):156, 1959.

[23] B. M. Evans and E. H. Chi. Towards a model of understanding social search. In *Proceedings of the 2008 ACM conference on Computer supported cooperative work*, CSCW '08, pages 485–494, New York, NY, USA, 2008. ACM.

[24] J. Feo, D. Harper, S. Kahan, and P. Konecny. Eldorado. In *Proc. 2nd conference on Computing frontiers*, CF '05, pages 28–34, New York, NY, USA, 2005. ACM.

[25] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[26] L. C. Freeman. Centrality in social networks: a conceptual clarification. *Social Networks*, 1(3):215 – 239, 1978-1979.

[27] D. Gregor and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing*, POOSC, July 2005.

[28] Apache hadoop, 2011. Available at `http://hadoop.apache.org/`.

[29] HipG: High-level distributed processing of large-scale graphs, July 2011. Available at `http://www.cs.vu.nl/~ekr/hipg/`.

[30] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2):303–325, 2011.

[31] E. Krepska, T. Kielmann, W. Fokkink, and H. Bal. A high-level framework for distributed processing of large-scale graphs. In *Proceedings of the 12th international conference on Distributed computing and networking*, ICDCN'11, pages 155–166, Berlin, Heidelberg, 2011. Springer-Verlag.

[32] V. Kumar, A. G. A. Gupta, and G. Karpis. *Introduction to Parallel Computing*. Addison Wesley, 2nd edition edition, 2003.

[33] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5:308–323, Sept. 1979.

[34] R. N. Lichtenwalter and N. V. Chawla. DisNet: A framework for distributed graph computation, 2011. To appear in proc. 2011 International Conference on Social Networks Analysis and Mining (ASONAM), Kaohsiung, Taiwan.

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

[36] K. Madduri and D. A. Bader. Compact graph representations and parallel connectivity algorithms for massive dynamic network analysis. In *Proc. Int. Parallel and Distributed Processing Symposium*, IPDPS, pages 1–11, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[37] K. Madduri and D. A. Bader. Small-world Network Analysis and Partitioning–Version 0.4, Aug. 2010. Available at `http://snap-graph.sourceforge.net/`.

[38] M. Magnani and L. Rossi. The ml-model for multi layer network analysis. In *IEEE International conference on Advances in Social Network Analysis and Mining*, 2011.

[39] M. Magnani, L. Rossi, and D. Montesi. Information propagation analysis in a social network site. In *2010 International Conference on Advances in Social Networks Analysis and Mining*, pages 296–300, LOS ALAMITOS – USA, 2010. IEEE computer Society.

[40] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard–Version 2.2, Sept. 2009. Available at `http://www.mpi-forum.org/docs/`.

[41] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1), Jan 1998.

[42] J. L. Moreno and H. H. Jennings. *Who shall survive? : a new approach to the problem of human interrelations*. Nervous and Mental Disease Publishing Co., Washington, D. C., 1934.

[43] OpenMP Architecture Review Board. OpenMP Application Program Interface–Version 3.1, July 2011. Available at `http://openmp.org/wp/`.

[44] T. Opsahl, F. Agneessens, and J. Skvoretz. Node centrality in weighted networks: Generalizing degree and shortest paths. *Social Networks*, 32(3):245–251, 2010.

[45] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford Digital Library Technologies Project, 1998.

[46] Project pegasus, July 2011. Available at `http://www.cs.cmu.edu/~pegasus/`.

[47] Sandia National Laboratories. Multi-Threaded Graph Library–Version 1.0, June 2011. Available at `https://software.sandia.gov/trac/mtgl`.

[48] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User guide and reference manual*. Addison-Wesley, Boston, MA, 2002.

[49] R. Trobec, M. Vajteršic, and P. Zinterhof, editors. *Parallel Computing: Numerics, Applications, and Trends*. Springer, 2009.

[50] D. J. Watts and S. H. Strogatz. Collective dynamics of “small-world” networks. *Nature*, 393(6684):440–2, 1998.

[51] J. Weng, E.-P. Lim, J. Jiang, and Q. He. Twitterrank: finding topic-sensitive influential twitterers. In *Proc. third ACM international conference on Web search and data mining*, WSDM ’10, pages 261–270, New York, NY, USA, 2010. ACM.

[52] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An api for programming with millions of lightweight threads. In *22nd IEEE International Symposium on Parallel and Distributed Processing*, IPDPS, pages 1–8, Miami, FL USA, Apr.14–18 2008. IEEE.

[53] D. White and S. Borgatti. Betweenness centrality measures for directed graphs. *Social Networks*, 16(4):335–346, Oct. 1994.