Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica

LARGE-SCALE NETWORK ANALYSIS ON DISTRIBUTED ARCHITECTURES

Tesi di Laurea in Algoritmi e Strutture Dati

Relatore:
Dott. Moreno Marzolla

Correlatore:
Dott. Matteo Magnani

Presentata da:
Carmine Paolino

Sessione III
Anno Accademico 2009/2010

To my parents

Sommario

Questa dissertazione esamina le sfide e i limiti che gli algoritmi di analisi di grafi incontrano in architetture distribuite costituite da personal computer. In particolare, analizza il comportamento dell'algoritmo del PageRank così come implementato in una popolare libreria C++ di analisi di grafi distribuiti, la Parallel Boost Graph Library (Parallel BGL).

I risultati qui presentati mostrano che il modello di programmazione parallela Bulk Synchronous Parallel è inadatto all'implementazione efficiente del PageRank su cluster costituiti da personal computer. L'implementazione analizzata ha infatti evidenziato una scalabilità negativa, il tempo di esecuzione dell'algoritmo aumenta linearmente in funzione del numero di processori.

Questi risultati sono stati ottenuti lanciando l'algoritmo del PageRank della Parallel BGL su un cluster di 43 PC dual-core con 2GB di RAM l'uno, usando diversi grafi scelti in modo da facilitare l'identificazione delle variabili che influenzano la scalabilità. Grafi rappresentanti modelli diversi hanno dato risultati differenti, mostrando che c'è una relazione tra il coefficiente di clustering e l'inclinazione della retta che rappresenta il tempo in funzione del numero di processori. Ad esempio, i grafi Erdős–Rényi, aventi un basso coefficiente di clustering, hanno rappresentato il caso peggiore nei test del PageRank, mentre i grafi Small-World, aventi un alto coefficiente di clustering, hanno rappresentato il caso migliore. Anche le dimensioni del grafo hanno mostrato un'influenza sul tempo di esecuzione particolarmente interessante. Infatti, si è mostrato che la relazione tra il numero di nodi e il numero di archi determina il tempo totale.

Introduction

The last 30 years have seen tremendous advances in microprocessor technology. Processors's clock rates have increased from about 8MHz (e.g., a Motorola 68000, circa 1979) to over 3.0GHz (e.g., an AMD Opteron, circa 2005). Keeping up with Moore's Law has become extremely challenging as chip-making technologies are approaching physical limits [Bor99].

In response, microprocessors manufacturers looked for other ways to improve performance. The first dual-core processors for personal computers were announced in 2005; today's multi-core processors represent the majority of the market. Their parallel nature, great computing power, and inexpensiveness make them a great fit for distributed platforms. In fact, as of November 2010 the 6 most powerful supercomputers of the world use widely available multi-core processors[1].

With respect to applications, simulations of physical events and signal processing dominated the field until one and a half decades ago. Since then, thanks to the massive growth of the Internet, web companies faced scalability problems due to the increasing number of people who started using their services. Thus, distributed computing became popular even among commercial applications.

Often those services require or produce massive amounts of data. The analysis of this data poses new challenges for distributed computing platforms, particularly when data is structured as a network. A network (or *graph*, in mathematics) is an abstract representation of a set of objects connected in pairs. Many objects of the real world can be thought of as networks,

[1]source: the 36th TOP500 List, released on November 14, 2010. `http://www.top500.org/lists/2010/11`

e.g., the Internet (a set of computers linked by data connections) and social networks (collections of people linked by their relationships). In fact, network analysis has many applications in fields like sociology, biology, physics, computer science, economics and operations research and often leads to interesting and useful insights.

This work looks at the challenges and the limits that network analysis algorithms face in distributed architectures constituted of commodity computers. In particular, analyzes the behavior of the PageRank algorithm as it is implemented in a popular C++ distributed graph library, the Parallel Boost Graph Library.

This dissertation is divided in 4 chapters. Chapter 1 introduces distributed computing. Chapter 2 presents an overview of networks and describes the algorithms to efficiently analyze them. Chapter 3 describes the PageRank algorithm as it is implemented in the Parallel Boost Graph Library. Finally, Chapter 4 presents an analysis of its limits and scalability challenges.

Contents

List of Figures

List of Tables

Chapter 1

Distributed Computing

In this era of rapid scientific and technological development, the need for fast processing of large amounts of data is greater than ever. Computationally intensive tasks that produce and/or consume huge datasets, like accurate simulations of physical events, are critical for the scientific world. Today, thanks to the massive adoption of the Internet, even commercial applications like databases and web applications need massive amount of computing power in order to scale to the number of clients and data that they have to support every day. The only practical, flexible and cost-effective solution to these problems is represented by distributed computing.

Typically, computer programs have been written for serial computation: algorithms are broken into serial streams of instructions, executed one after another in a single Central Processing Unit (CPU). Distributed computing, on the other hand, uses multiple CPUs simultaneously to solve a single problem. This is based on the principle that an algorithm can be broken into discrete parts to be executed concurrently, distributing the workload among all the computing resources, therefore using less time.

This definition is purposefully broad enough to include single computers with single processors with multiple cores, single computers with multiple processors, multiple computers connected through custom networks, multiple computers connected through the internet, or any combination of the above.

In the past few years, the semiconductor industry switched its focus from

increasing clock frequencies to integrating multiple cores into the same package. This and the inexpensiveness of today's desktop computers, determined the current success of distributed computing.

1.1 History

The concept of running multiple programs at once is not new. In the early 1960s, the invention of independent device controllers made it possible to run a new or suspended program while I/O operations were performed by the controller. In this scenario, the CPU commands the controller to read or write a portion of disk. This operation terminates immediately, giving the CPU the possibility to run another program. When the controller finishes, it issues an hardware interrupt that causes the processor to save its state and begin the execution of an interrupt handler. Therefore, concurrency was of concern of operating systems designers.

In 1962, Burroughs introduced the first computer with multiple identical CPUs connected to a single shared main memory, the D825 [And00]. This created a challenge for both operating system designers and application programmers and led to a flood of papers in the 1970s about formal methods of parallel computation.

The late 1970s and early 1980s have seen the introduction of computer networks. Networks made easy to connect multiple computers and therefore marked the rise of distributed programming which quickly became a major topic in the 1980s and even more in the 1990s.

The pervasiveness of multicore CPUs and fast computer networks coupled with physical problems faced by chip manufacturers, show that distributed computing is the future of computing.

1.2 Terminology

1.2.1 The von Neumann Architecture

John von Neumann, mathematician and early computer scientist, published a series of papers in 1945 [vNG93] where he proposed a model for a stored-program computer, a computer that keeps data and instructions, coded with specific symbols, in the same memory.

At the time, when computers were still programmed by setting switches and inserting patch cords, this architecture was revolutionary and quickly became very popular. Virtually all computers followed this design since then.

Still, it is a very generic and simple architecture including 5 parts: Memory, Control Unit, Arithmetic Logic Unit, Input and Output organized as in Figure 1.1.

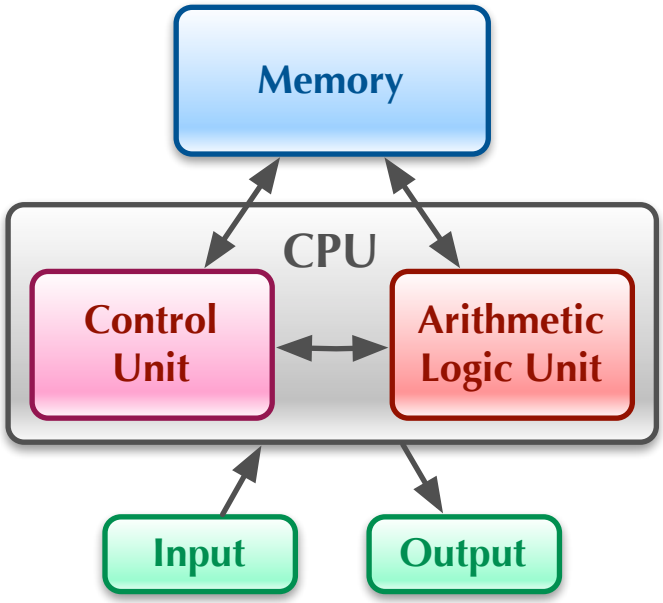


Figure 1.1: Schematic of the von Neumann architecture

Memory is a computer part in which data and encoded program instructions can be stored for retrieval. It actually represents the system's entire memory hierarchy, from extremely fast, expensive and small caches to slow,

cheap and large tapes for backups. The *Arithmetic Logic Unit (ALU)* performs arithmetic and logical operations. The *Control Unit* is the part of the CPU that coordinates the work that needs to be done in order to accomplish the programmed task. It fetches and decodes the instructions stored in memory, commands the ALU to execute them, reads and writes data from memory, and activates the resources needed. Finally, *Input/Output (I/O)* devices are interfaces to the user, like monitors (output) and keyboards (input).

1.2.2 Flynn's Taxonomy

In 1972 Michael J. Flynn, in his paper *Some Computer Organizations and Their Effectiveness* [Fly72], proposed a taxonomy of parallel computer systems based on the number of concurrent instructions and data streams available in the architecture. It is organized in two independent dimensions: **instruction** and **data**, each of which can be either **single** or **multiple**. Thus, the four possible Flynn's classifications are:

	Single Data	Multiple Data
Single Instruction	**SISD**	**SIMD**
Multiple Instruction	**MISD**	**MIMD**

Table 1.1: Matrix of the Flynn's taxonomy

SISD: Single Instruction, Single Data

As the name implies, the **Single Instruction, Single Data** class is composed by uniprocessors systems that can only execute a single instruction stream that operates on a single data stream at a time. This corresponds to the von Neumann architecture (see Subsection 1.2.1) and represents the longest standing type of computers. However, even if the **SISD** architecture describes inherently serial computers, concurrency can be achieved with implicit parallelism techniques such as pipelining, that allows overlapping execution of multiple instructions within the same circuitry.

SIMD: Single Instruction, Multiple Data

The **SIMD** architecture describes computers with multiple processors that perform the same operation on multiple data simultaneously, exploiting data parallelism. They first appeared in the 1970s in vector supercomputers which were eventually replaced by inexpensive scalar **MIMD** clusters. When personal computers became common and powerful enough to support real-time gaming, chip manufacturers turned to **SIMD** to meet the demand of this particular type of computation. Now, every personal computer has a CPU that supports a **SIMD** instructions set and at least one dedicated **SIMD** processor for graphics, called Graphics Processing Unit (GPU).

MISD: Multiple Instruction, Single Data

The **MISD** architecture describes computers with multiple processors that perform different operations on the same data. Very few instances of this architecture exist, because **SIMD** and **MIMD** are often more appropriate for common parallelism techniques. However, fault-tolerant computers can benefit from this architecture by executing the same instructions redundantly on many processors in order to detect and correct errors.

MIMD: Multiple Instruction, Multiple Data

A **MIMD** computer system is comprised of multiple processors that can execute different instructions on different data streams, asynchronously and independently. The **MIMD** architecture is the most common type of parallel computer today, thanks to the massive adoption of multicore and multiprocessor architectures for personal computers.

1.3 Memory Architectures

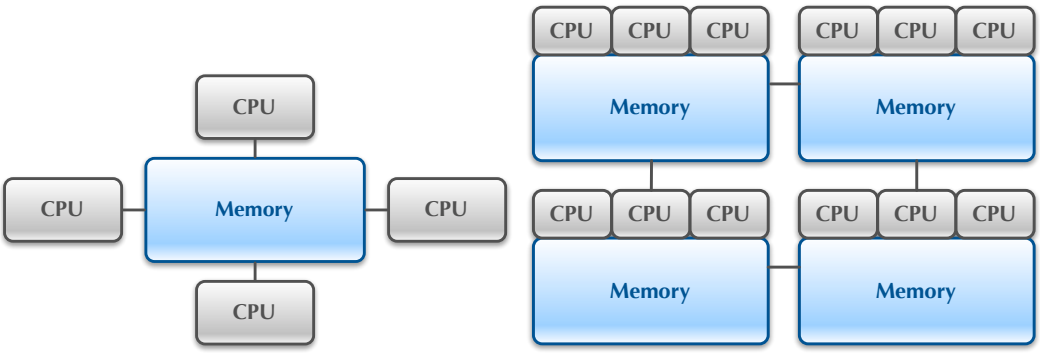
Managing memory is by far the most troubling part of programming parallel computers, and different memory architectures yield different challenges.

There are two basic memory architectures for parallel computers: shared memory and distributed memory.

1.3.1 Shared Memory

In a shared-memory multiprocessor system, memory's global address space is accessible to all processors. Processors interact by modifying data in this memory and changes are instantly visible to all of them.

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



(a) UMA (b) NUMA

Figure 1.2: Schematics of Shared Memory architectures

Small multiprocessor systems typically have a single connection between a processor and the memory, therefore memory access times are the same across all processors. Such systems are called UMA machines. Figure 1.2a shows an example of a UMA machine with 4 processors.

In large shared memory multiprocessors, those having tens or hundreds of processors, each processor has local and non-local (local to another processor) memory. This is the cause of non-uniform memory access times, in fact such systems are called NUMA machines. Figure 1.2b shows an example schematic of a NUMA machine with 12 processors and 4 memories.

Thanks to the global memory space, Shared Memory machines are much easier to program than other parallel systems. From the programmer point of view, read operations are identical to serial programs and can be executed at any time without any lock mechanism. Only write operations are trickier

because they require mutual exclusion for concurrent processes.

1.3.2 Distributed Memory

In contrast to the shared memory model, the distributed memory model defines computers in which each segment of the memory is physically associated with a different processor. Because each processor can only address the local memory, communication is achieved through message passing via an interconnection network, like in Figure 1.3.

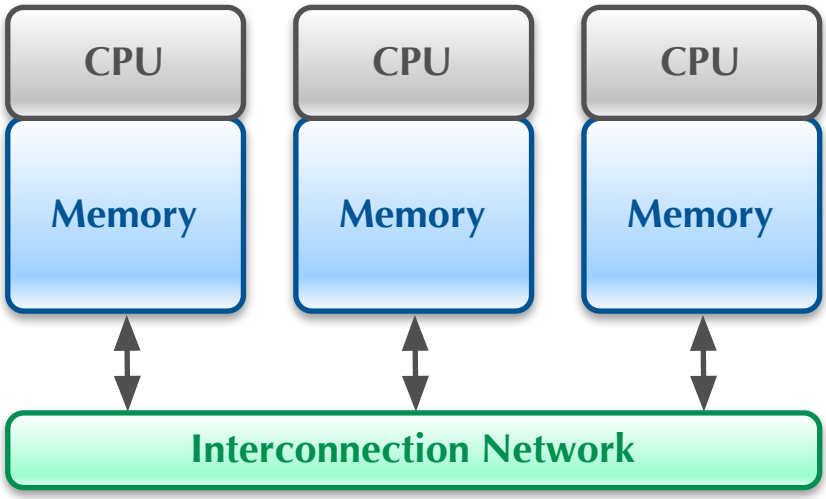


Figure 1.3: Schematic of the Distributed Memory architecture

Distributed memory systems are significantly harder to program than shared memory ones. Programmers are responsible for distributing the data among all the processors, in a way that communication is reduced to a minimum.

1.3.3 Hybrid Distributed-Shared Memory

Today, the most powerful supercomputers make use of the hybrid distributed-shared memory model and current trends indicate that this will continue to be the case in the future. Machines of this category are generally shared memory computers connected through a network, like in Figure 1.4.

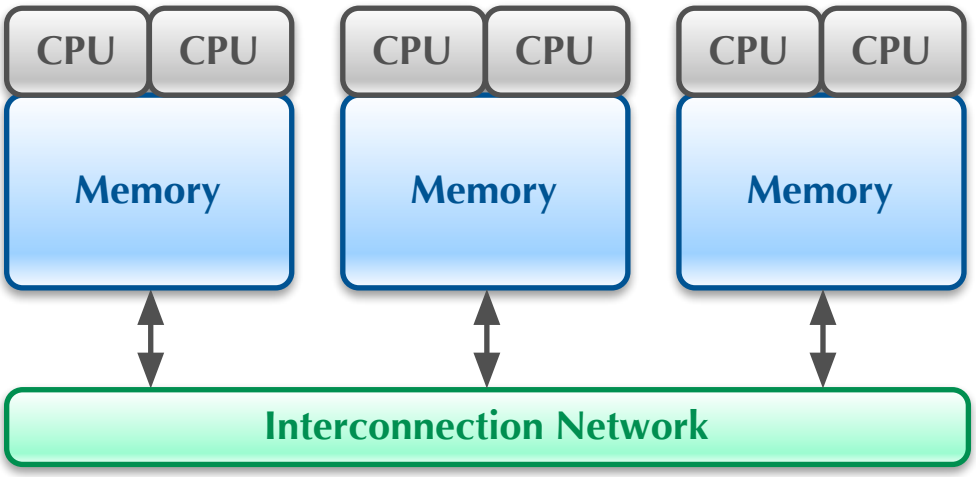


Figure 1.4: Schematic of the Distributed-Shared Memory architecture

Challenges in programming this type of parallel computer are the union of challenges of both shared-memory and distributed-memory architectures.

1.4 Programming models

Programming models are abstractions above hardware and memory architectures, thus they can be implemented on virtually any machine. The most important widely-used parallel programming models are threads and message passing.

1.4.1 Threads

In the threads programming model, each process can split into multiple concurrent executable programs. Threads are relatively easier to implement in uniprocessor and shared-memory architectures and are the smallest unit of processing that can be scheduled by an operating system.

Threads implementations typically comprises libraries of routines called by parallel code, such as POSIX Threads, or compiler directives embedded in serial code, such as OpenMP. POSIX Threads-like implementations require the programmer to explicitly call the parallel primitives, giving him all

the power but also all the responsibility. OpenMP (`http://openmp.org/`) instead provides compiler directives that can be embedded in serial code to automatically parallelize it. It is simpler to use and potentially more effective for inexperienced parallel programmers because it uses well-tested, efficient techniques to parallelize serial code.

1.4.2 Message Passing

In the message passing model of distributed computing, processes can send and receive messages from other processes. Each process uses its own local memory during computation and communicates to other processes that can reside on the same physical machine and/or across an arbitrary number of machines, interconnected by a network. Typically send and receive operations must be cooperative, e.g., a send operation must have a matching receive operation.

Implementations of the message passing model comprise libraries of routines that implement the communication protocols and techniques. The programmer is then responsible for determining all parallelism explicitly. A common interface for message passing implementations have been standardized by the the MPI Forum in 1994. This interface, called Message Passing Interface (MPI), is now the de facto standard for message passing, used in virtually every message passing implementation out there. MPI is typically used for distributed memory architectures and can be combined with the thread model to exploit the characteristics of hybrid distributed-shared memory architectures. The latest version of MPI is 2.2, approved September 4, 2009 by the MPI Forum and available at `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`.

Bulk Synchronous Parallel

The Bulk Synchronous Parallel (BSP) is a theoretical model for designing parallel algorithms. It consists of processors with fast local memories connected by a communication network. Communications happen as in the message passing model. However, a BSP computation proceeds in a series of

global *supersteps*. A superstep consists of three ordered stages:

1. *computation*: each processor computes independently from other processes, using only values stored in the local memory;
2. *communication*: processes exchange data;
3. *barrier*: a process that reaches the barrier waits until all other processes have completed their communication actions.

Figure 1.5 shows a diagram of a BSP superstep.

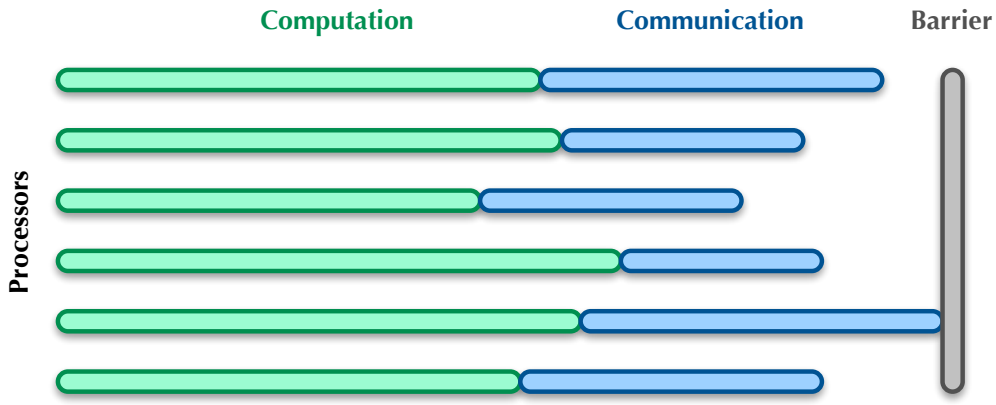


Figure 1.5: Diagram of a Bulk Synchronous Parallel superstep

Chapter 2

Networks

Many systems of interest to scientists are composed of individual parts or components linked together in some way. Examples range from the Internet (a set of computers linked by data connections) to social networks (collections of people linked by their relationships). While there are fields that study the nature of the individual components, for instance how a computer works or how a human being feels or acts, and others that study the nature of the connections or interactions, like the communication protocols or the dynamics of human relationships, there is a field that studies the *patterns* of connections between components. That field is called Network Analysis.

2.1 Terminology

A *network*, or *graph*, is a mathematical structure used to model pairwise relations between objects. Objects are called *nodes* or *vertices* and relations that link them are called *edges* or *ties*. Throughout this dissertation I will denote the number of nodes by n and the number of edges by m. A small network consisting of $n = 8$ nodes and $m = 9$ edges is shown in Figure 2.1.

In order to model asymmetric relations, edges can have directions; in this case we call the graph *directed*, otherwise it is called *undirected*. A node is *adjacent* to another node if the two are directly connected by an edge. A node j is *reachable* from node i if exists a sequence of adjacent

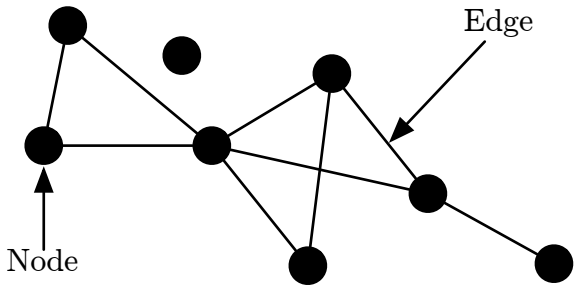


Figure 2.1: A small network with $n = 8$ nodes and $m = 9$ edges

nodes, called *path*, from node i to node j. A graph is *connected* if every node is reachable from some other node; e.g., the graph in Figure 2.1 is not connected, because it contains an unreachable node. A *cycle* is a path which starts and finishes in the same node. A graph is *acyclic* if doesn't contain any cycles. The *length* of a path is the number of edges traversed along the path. The *geodesic distance* between two nodes is the number of edges in the shortest path connecting them. The *characteristic path length* of a graph is the average geodesic distance between all pairs of nodes. A *subgraph* is a connected portion of a graph. A *clique* is a subgraph in which all nodes are adjacent to all others. Nodes and edges can be labeled with additional informations such as names or weights. A *weighted* network is a network in which nodes have some form of weight attached to them.

2.1.1 Clustering coefficient

The clustering coefficient is a measure that assesses the degree to which nodes tend to cluster together. It is defined as

$$C = \frac{3 \times (\text{number of triangles})}{(\text{number of connected triples})}, \tag{2.1}$$

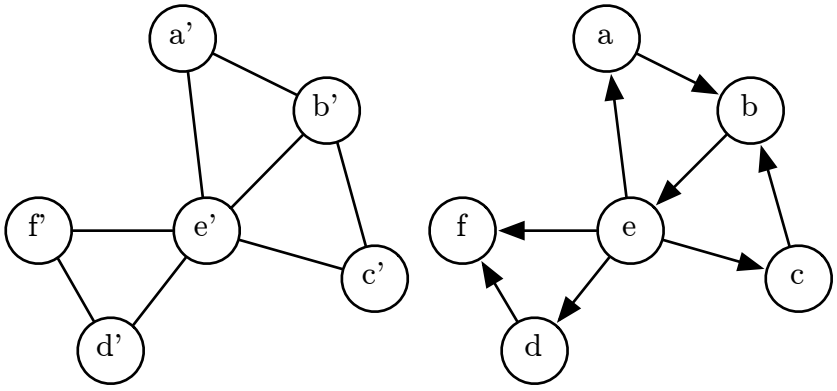
where a *triangle* is a clique composed of three nodes, and a *triple* is a subgraph with three nodes.

The clustering coefficient is particularly interesting for studying how communication flows in a network. For example, infectious diseases spread more

easily in small-world networks, and generally in networks with high clustering coefficients, than in other network types [WS98].

2.2 Centrality measures

Centrality is an important structural attribute of networks because it determines the relative importance of a vertex within the graph. Since it is not a clearly defined concept, not even within a single type of network [Fre79], a number of centrality measures have been created. The four most important ones are degree, betweenness, closeness and eigenvector centrality. However, for the purpose of this work I will present only an overview of the degree centrality, its generalization to weighted networks, and the PageRank. This section will adopt Figure 2.2a as the example undirected network and Figure 2.2b as the example directed network.



(a) Undirected (b) Directed

Figure 2.2: Two small networks with six nodes and eight edges

2.2.1 Degree Centrality

The simplest centrality measure is, perhaps, degree centrality. It matches the *degree* of a node, defined as the number of adjacent nodes. For an undi-

rected graph if N is its node set, the degree k_i of node $i \in N$ is:

$$k_i = \sum_{j \in N} A_{i,j}, \quad \text{where} \quad A_{i,j} = \begin{cases} 1, & \text{if } i \text{ is adjacent to } j; \\ 0, & \text{otherwise.} \end{cases} \tag{2.2}$$

In a directed graph each node has two degrees. If $i \in N$ is a node, its *in-degree* k_i^{in} is the number of edges where i is the destination and the *out-degree* k_i^{out} is the number of edges where i is the source. Formally:

$$k_i^{\text{in}} = \sum_{j \in N} A_{j,i} \quad \text{and} \quad k_i^{\text{out}} = \sum_{j \in N} A_{i,j}. \tag{2.3}$$

Table 2.1a and Table 2.1b show, respectively, the degree centralities for the undirected graph in Figure 2.2a and the directed graph in Figure 2.2b.

Node	Degree
a'	2
b'	3
c'	2
d'	2
e'	5
f'	2

(a) Undirected

Node	In-Degree	Out-Degree
a	1	1
b	2	1
c	1	1
d	1	1
e	1	4
f	2	0

(b) Directed

Table 2.1: Degree centralities for the example graphs

2.2.2 Node strength

In 2010, Opsahl et al. [OAS10] proposed a generalization of degree centrality to weighted networks called *node strength*. The node strength s_i of node $i \in N$ is defined as

$$s_i = \sum_{j \in N} w_{i,j}, \tag{2.4}$$

where $w_{i,j}$ is the weight of the edge that goes from i to j, or 0 if the edge doesn't exist.

When $w_{i,j} = 1 \; \forall i, j \in N$, node strength is equivalent to degree centrality.

2.2.3 PageRank

In 1998, Larry Page, Sergey Brin, Rajeev Motwani and Terry Winograd described a new algorithm for ranking Web pages, the PageRank [PBMW98]. Named after Larry Page[1] and used by the Google search engine, it was born as a method to measure human interest and attention to Web pages, but it can be applied to any directed network and is especially useful in citation-based networks.

The simplified PageRank algorithm models a random web surfer. It corresponds to the probability distribution of a random walk on the graph of the Web: the likelihood that a person randomly clicking on links will arrive at any particular page. The simplified PageRank R of a node i is defined as

$$R(i) = \sum_{j \in B_i} \frac{R(j)}{k_j^{\text{out}}}, \tag{2.5}$$

where B_i is the set of nodes that have edges going to i.

However, the random surfer will eventually "get bored" and request another page. The probability of this event, at any step, is called *damping factor*. The final version of the PageRank PR, as found in [BP98], is then

$$PR(i) = (1 - d) + d \sum_{j \in B_i} \frac{PR(j)}{k_j^{\text{out}}}, \tag{2.6}$$

where d is the damping factor. This definition led to some confusion, since the same paper states that "the sum of all Web pages's PageRanks will be one", which it can be proven that it is not the case with this formula. So the normalized PageRank PR^*, actually used in Google, is

$$PR^*(i) = \frac{1 - d}{n} + d \sum_{j \in B_i} \frac{PR^*(j)}{k_j^{\text{out}}}. \tag{2.7}$$

[1] source: `http://www.google.com/press/funfacts.html`

A comparison between the original PageRank (Equation 2.6) and the normalized PageRank (Equation 2.7) for the graph in Figure 2.2b is shown in Table 2.2.

Node	Original PageRank (PR)	Normalized PageRank (PR^*)
a	0.3011556	0.09906545
b	0.6612685	0.23316367
c	0.3011556	0.09906545
d	0.3011556	0.09906545
e	0.7117302	0.26906185
f	0.5567899	0.20057812

Table 2.2: PageRank values for the graph in Figure 2.2b

2.3 Models

2.3.1 Erdős–Rényi

Erdős–Rényi is a graph model proposed in 1959 [ER59] by Hungarian mathematicians Paul Erdős and Alfréd Rényi. It is used to generate pure-random graphs that are constructed by connecting nodes randomly and independently. In an Erdős–Rényi graph $G_{n,p}$ with n nodes, each pair of nodes is connected with probability p. The expected number of edges in such graph is $\binom{n}{2}p$, and its clustering coefficient is p. Figure 2.3 shows an Erdős–Rényi graph $G_{20,0.1}$.

Since every edge is constructed independently from other edges, Erdős–Rényi graphs typically show little structure, therefore they are rarely used to represent real-world networks. Nevertheless, they are particularly useful in testing graph algorithms's performance.

2.3.2 Small-World

Small-World networks, defined in 1998 by Duncan J. Watts and Steven H. Strogatz [WS98], model many real-world networks, such as neural networks,

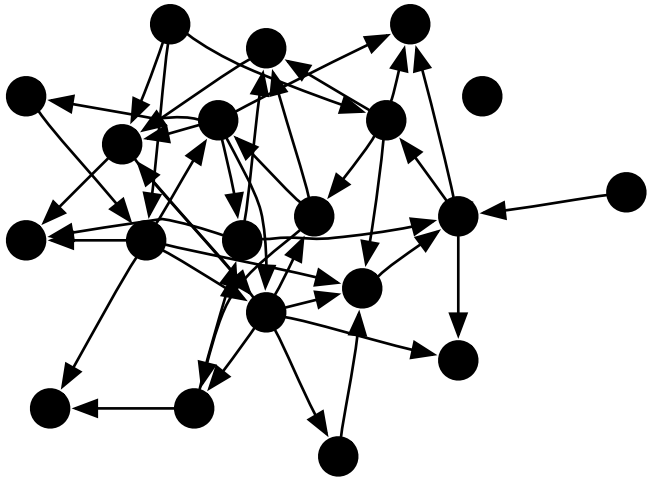


Figure 2.3: An example Erdős–Rényi graph $G_{20,0.1}$

social networks, power grids, the Internet, and other self-organizing systems. These systems show high clustering coefficients and short characteristic path length. Specifically, in Small-World networks the characteristic path length L is proportional to the node count of the graph:

$$L \propto \log(n) \tag{2.8}$$

Watts and Strogatz also proposed the following algorithm to generate Small-World networks. To construct a Small-World graph $W_{n,k,\beta}$, where n is the total number of nodes, k is the mean degree $n \gg k \gg \ln(n) \gg 1$, and $0 \leq \beta \leq 1$ is the rewire probability, the algorithm:

1. arranges the n nodes as a ring, connecting each node to k neighbors, $k/2$ on each side;

2. rewires every edge with probability β, keeping the source and choosing the destination between all possible nodes that are not the source.

Figure 2.4 shows an example Small-World graph $W_{20,4,0.05}$.

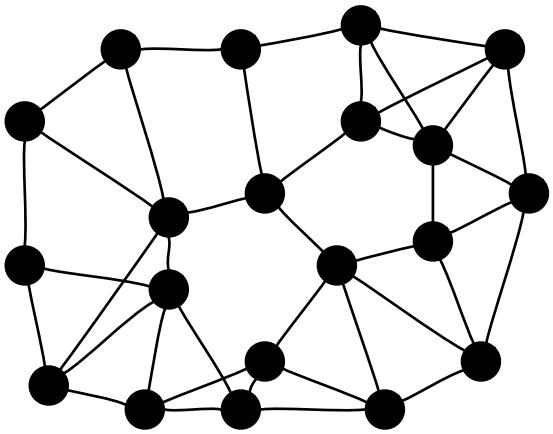


Figure 2.4: An example Small-World graph $W_{20,4,0.05}$

Chapter 3

An Overview of the Parallel Boost Graph Library

The Parallel Boost Graph Library (Parallel BGL) [GL05] is a library for distributed graph computation. It is part of Boost (`http://www.boost.org/`), a popular collection of open source peer-reviewed libraries that extend the C++ language's functionalities. The Parallel BGL builds on the Boost Graph Library (BGL) [SLL02], a library for sequential graph computation also part of Boost, offering similar data structures, algorithms, and syntax. Being primarily concerned with distributed graphs, the Parallel BGL contains distributed graph data structures, distributed graph algorithms and abstractions over the communication medium.

A distributed graph is a graph stored on a distributed system, where nodes and edges are spread across multiple processes. Each process stores a subset of the total nodes and the edges that connect them either locally (both endpoints are stored on the same process) or remotely (one of the endpoint is stored on a different process).

The Parallel BGL is built around the generic programming paradigm. This paradigm requires algorithms to be as general as possible, making minimal assumptions about data types, but be just as efficient as the original algorithm when instantiated in a concrete case. These characteristics make the Parallel BGL efficient and flexible in terms of graph, node, and edge data

structures.

The Parallel BGL is intended primarily as a research platform, to facilitate both experimentation and comparison of parallel graph algorithms and to provide solid implementations for solving large-scale graph problems.

3.1 Distributed Data Structures

Parallel BGL's generic algorithms and the flexibility offered by property maps (introduced in Subsection 3.1.2) make it possible to reuse existing compatible data structures. However, in order to be useful out of the box, the library also provides a distributed graph data structure: the distributed adjacency list.

3.1.1 Distributed Adjacency List

The adjacency list is a data structure that represents graph with l lists, one for each node i, that store i's neighbors. The distributed adjacency list extends this concept by assigning different subsets of those lists to different processes.

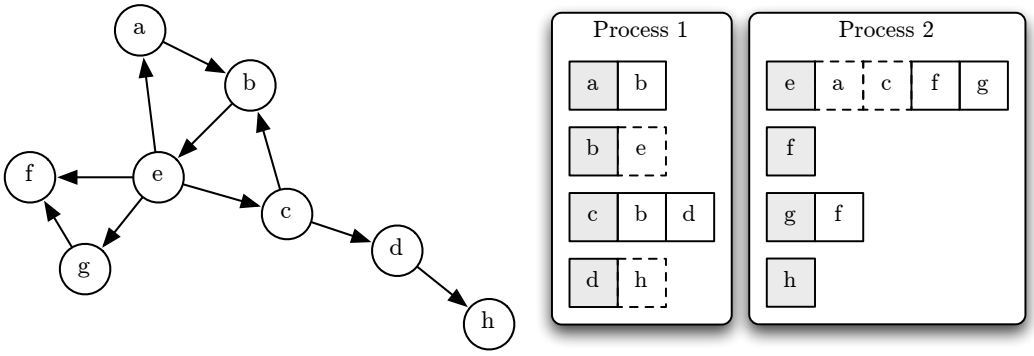
Figure 3.1a shows an example graph with eight nodes and ten edges. Figure 3.1b shows the same graph stored on two processes as a distributed adjacency list where gray squares represent nodes, white squares represent their neighbors and dashed white squares represent adjacent nodes stored on a remote process.

The Parallel BGL's implementation of the distributed adjacency list can represent undirected, directed and bidirectional graphs[1] and can use different distribution types to partition nodes in graphs.

3.1.2 Distributed Property Map

The Boost Property Map library provides a general purpose interface for mapping key objects to corresponding value objects, hiding the details of

[1] a bidirectional graph is a graph that stores both the in-edges and the out-edges in each node, in contrast to the directed graph which stores only the out-edges.



(a) Conceptual (b) Distributed Adjacency List

Figure 3.1: Comparing two representations of the same graph

how the mapping is implemented. The Parallel BGL builds on this concept by distributing property values across multiple processors and by providing transparent access to local and remote values. However, additional care is required when dealing with distributed property maps. Write operations (implemented in the `put` function) that modify remote property values are cached in local *ghost cells* before the communication occurs and can be rejected by the remote process. Moreover, read operations (implemented in the `get` function) can return outdated values because they only read the content of local *ghost cells*, updated at every `synchronize` call. To address these issues, the Parallel BGL provides reduction operations and consistency models.

Reduction operations

Reduction operations maintain consistency by managing default values and multiple writes. They are implemented as structures that contains two functions: one that determines a default value when a remote value is not immediately available, and another one called when a value is received from another process, in order to determine which one will be stored in the local property map between the local value, the remote value, or some combination of both.

PageRank's reduction operation, called `rank_accumulate_reducer`, returns 0 when a value is not available and the sum of the local and remote value when a value is received from another process. In other words, it specifies the default rank for a remote node to be 0. When a remote rank is received, it accumulates it by adding the remote one to the locally cached one. With this setup the property map itself will compute the partial sums on each processor and then accumulate the results on the owning processor after each `synchronize` call.

Consistency models

Consistency models are flags that define how values propagate across processes. The Parallel BGL contains six consistency models, but for the purpose of this dissertation the most interesting ones are `cm_flush` and `cm_reset`.

`cm_flush` instructs the distributed property map to queue all the remote write operations in local ghost cells, and then flush them to the owning processors after each `synchronize` call. Once the flush has occurred, `cm_reset` resets the local ghost cells to their default value as determined by the reduction operation.

This combination of consistency models is used by the Parallel BGL's PageRank implementation to locally accumulate ranks at each step.

3.2 PageRank implementation

The Parallel BGL's PageRank implementation is based on the original PageRank algorithm as defined is Equation 2.6 and implements the Bulk Synchronous Parallel model of computation. Boost 1.46.0[2] defines three `page_rank` functions in `<boost/graph/page_rank.hpp>` in order to provide a rich interface to the actual PageRank implementation. These functions are, in descending order of generality:

1. `page_rank(Graph, RankMap);`

[2]released February 21, 2011 and available at `http://sourceforge.net/projects/boost/files/boost/1.46.0/`

2. `page_rank(Graph, RankMap, Done);`

3. `page_rank(Graph, RankMap, Done, Damping, NumVertices);`

where:

- `Graph` is the graph to analyze;
- `RankMap` is a property map that maps each node to its rank;
- `Done` is a functor that determines when the PageRank algorithm should complete; the first `page_rank` function sets it to `n_iterations(20)` (which will make the PageRank algorithm run for 20 iterations) and calls the second `page_rank` function;
- `Damping` is the damping factor; the second `page_rank` function sets it to 0.85 and calls the third `page_rank` function;
- `NumVertices` is the number of nodes stored in the current process.

Inevitably, then, the program reaches the third `page_rank` function. This function passes all the aforementioned values to `page_rank_impl`, defined in `<boost/graph/distributed/page_rank.hpp>`.

First, `page_rank_impl` sets up the computation by assigning the rank of all `NumVertices` nodes to 1/`NumVertices`, applying the `cm_flush` and `cm_reset` consistency models, and designating `rank_accumulate_reducer` as the reduction operation. Then, `page_rank_impl` loops until `Done` returns `true`, calling `page_rank_step` and `synchronize` at each iteration. Here, `page_rank_step` implements the computation phase of the BSP and `synchronize` both the communication phase and the barrier.

`page_rank_step` is a function defined in `<boost/graph/page_rank.hpp>`. For all the nodes in the current process $i \in N_{\text{curr}}$: sets their ranks to $(1 -$ `Damping`$)$, and for all the nodes j adjacent to i, sums the current rank with $d\frac{PR(i)}{k_j^{\text{out}}}$. At the end of each step, the `synchronize` function ensures that every node receives and accumulates the ranks calculated in other processes.

Parallel BGL's authors Douglas Gregor and Andrew Lumsdaine claim that each step of this PageRank implementation requires $O(n + m)/p$ time on p processors and performs $O(n)$ communications.

Chapter 4

PageRank Performance Analysis

4.1 Terminology

The *wall clock time* is the actual duration of a task, measured by the time elapsed from the start to the end. The wall clock time includes I/O time, CPU time, communication delays, etc. In other words it's the difference between the time at which the task finished and the time at which the task started.

The *speedup* of a distributed program measures how much it is faster than the corresponding sequential program. It is defined by the following formula:

$$S_p = \frac{T_1}{T_p} \tag{4.1}$$

where p is the number of processors, T_1 is the wall clock time of the sequential program, and T_p is the wall clock time of the distributed program with p processors. When $S_p = p$, the distributed program has a *linear* (or *ideal*) speedup.

Due to the size of some graphs used in my tests and our parallel computer's memory constraints, I was unable to run sequential tests. Thus, the necessity of a modified speedup measure arose.

Definition 1 (nth speedup)**.** The nth speedup of a distributed program measures how much it is faster than the corresponding program when run on n processors. It is defined as

$$S_p^n = \frac{nT_n}{T_p}, \tag{4.2}$$

where p is the number of processors, T_n is the wall clock time of the distributed program run on n processors, T_p is the wall clock time of the distributed program with p processors. n is the number of processors used for the initial measurement.

Proof.

$$\begin{aligned} S_p^n &= \frac{nT_n}{T_p} \\ &= \frac{n\frac{T_1}{n}}{T_p} \\ &= \frac{T_1}{T_p} \end{aligned}$$

□

Corollary 2. When $S_p^n = p$, the distributed program has a linear speedup.

4.2 Methodology

I assessed the scalability of the PageRank implementation from the Parallel BGL by performing tests on a cluster of commodity computers. That cluster was composed of 43 PCs, each one with a dual-core Intel® Core™ 2 Duo E7500 processor running at 2.93GHz and 2GB of RAM, connected to one of the two 24-ports hubs in a star topology where each hub was then connected to a common switch. The lab's network topology is shown in Figure 4.1.

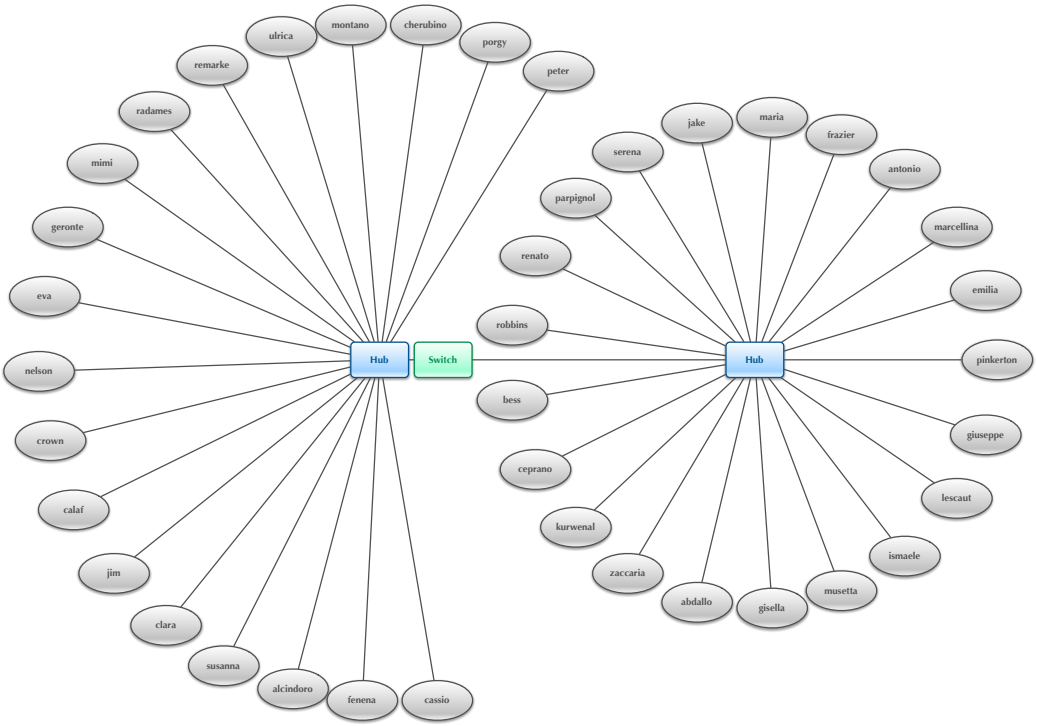


Figure 4.1: Our cluster's network topology. Gray ellipses represent machines, blue rectangles hubs, and green rectangles switches.

Since the lab was shared with other students I had to be very careful choosing the machines for each test. For that reason I developed three Ruby scripts that run the tests on idle machines (those with the lowest load average, which typically was 0.00) automatically increasing the number of processors used at each run and plot the time graphs at the end:

1. `list_csunibo.rb` lists all the cluster's computers;

2. `idle_servers.rb` selects idle machines so that the total number of available processors is equal or greater than the number of processors requested by the user (e.g., in a cluster of dual-core machines, if the user requests 30 idle processors, this script will return 15 machines with the lowest load average);

3. `plot_timegraphs.rb` glue the aforementioned scripts together, executes the tests on an increasing number of processors and plots the

time graphs.

The list of idle machines was computed at the start of the whole testing process, and at each run passed unchanged to `mpirun`. That way, `mpirun` was able to load balance the processors, spreading the processes among the highest possible number of processors. (For example, in a cluster of 10 dual-core machines, `mpirun` will assign one process per machine if the number of processes is less or equal than 10, and it will eventually use all the available 20 processors when the number of processes is 20.)

The wall clock time of the `page_rank` function, averaged out over all processes, was logged in a file with the following structure:

```
Algorithm GraphType NumberOfProcesses HumanReadableTime Seconds
```

For example, one of the actual log files used in this work looks like this:

```
PageRank CsvBigraph 4 00:01:28.663179 88.6632
PageRank CsvBigraph 6 00:01:44.027264 104.027
PageRank CsvBigraph 8 00:02:09.503521 129.504
PageRank CsvBigraph 10 00:02:27.669325 147.669
PageRank CsvBigraph 12 00:02:44.806108 164.806
PageRank CsvBigraph 14 00:02:56.309223 176.309
PageRank CsvBigraph 16 00:03:24.311819 204.312
PageRank CsvBigraph 18 00:03:31.438518 211.439
PageRank CsvBigraph 20 00:04:03.072149 243.072
PageRank CsvBigraph 22 00:04:17.186015 257.186
PageRank CsvBigraph 24 00:04:40.137390 280.137
PageRank CsvBigraph 26 00:04:47.866493 287.866
PageRank CsvBigraph 28 00:05:21.858589 321.859
PageRank CsvBigraph 30 00:05:53.098353 353.098
```

Results were grouped by graph type, node count, and edge count in order to spot patterns easily.

The source code of my tests is available as part of the LANA project (**La**rge-scale **N**etwork **A**nalyzer), available at `http://sigsna.trac.cs.unibo.it/` under the MIT license.

4.2.1 Datasets

Different datasets were chosen in order to study the impact of several variables that can influence the algorithms's wall clock time. Three network types were chosen for their different structural characteristics: Erdős–Rényi and Small-World networks synthesized by Parallel BGL's respective generators, and FriendFeed's users network. To ensure the repeatability of the tests, the random number generator used by Parallel BGL's generators was seeded with the same hard-coded number.

FriendFeed (`http://friendfeed.com/`) is a popular microblogging and social network service that makes public content easy to access. Thus, members of the SIGSNA project (Special Interest Group on Social Network Analysis, `http://larica.uniurb.it/sigsna/`), were able to scrape, extract, and reprocess FriendFeed's data [CDLM+10] producing a series of CSV (comma separated value) files available at `http://larica.uniurb.it/sigsna/data/` that cover all the entries and the active users between August 1, 2010 and September 30, 2010. For my purposes, only the followers's network[1] (where accounts are nodes and subscriptions to other account's feeds are edges) and its reduction to Italian users were considered.

The node and the edge count of FriendFeed's Global and Italian networks indirectly determined the size of some of the synthetic networks. Since the goal is understanding what and how much variables influenced the algorithm's wall clock time, the synthetic networks's sizes match FriendFeed networks's sizes as much as possible. Starting from this baseline, the node and the edge count of each synthetic network was increased independently from each other, resulting in the datasets shown in Table 4.1.

[1] available at `http://dbis.cs.unibo.it/sigsna/data/2010a/subscriptions.zip`

Graph Type	Nodes	Edges
FriendFeed Global	653646	27811816
FriendFeed Italian	28250	692668
Erdős–Rényi	28250	141250
Erdős–Rényi	28250	1412500
Erdős–Rényi	282500	1412500
Erdős–Rényi	282500	14125000
Erdős–Rényi	653646	27811816
Erdős–Rényi	1307292	55623632
Small-World	28250	141250
Small-World	28250	1412500
Small-World	282500	1412500
Small-World	282500	14125000
Small-World	653646	28106778
Small-World	1307292	56213556

Table 4.1: Datasets used for the tests

4.3 Results

4.3.1 Negative scalability

Figure 4.2 shows the wall clock time of the distributed PageRank implementation from the Parallel BGL on FriendFeed's Global and Italian networks as a function of the number of processors. Just by looking at this graph, it is clear that the wall clock time of the algorithm increases linearly with the number of processors or, in other words, it has *negative scalability*. In fact, the Pearson's correlation coefficient[2] between the number of processors p and the wall clock time for the Italian FriendFeed network t_I is $\rho_{p,t_I} = 0.9808243$, and for the Global FriendFeed network t_G is $\rho_{p,t_G} = 0.9966577$.

This relationship holds just as well with graphs different than FriendFeed's network. In fact, Figure 4.3 and Figure 4.4 feature the same pattern for Erdős–Rényi and Small-World graphs, respectively. Also, Table 4.2 shows

[2]The Pearson's correlation coefficient $\rho_{X,Y}$ is a measure of the linear dependence between two variables X and Y, giving a value between $+1$ (positive correlation) and -1 (negative correlation) with 0 indicating no correlation. It is defined as $\rho_{X,Y} = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y}$.

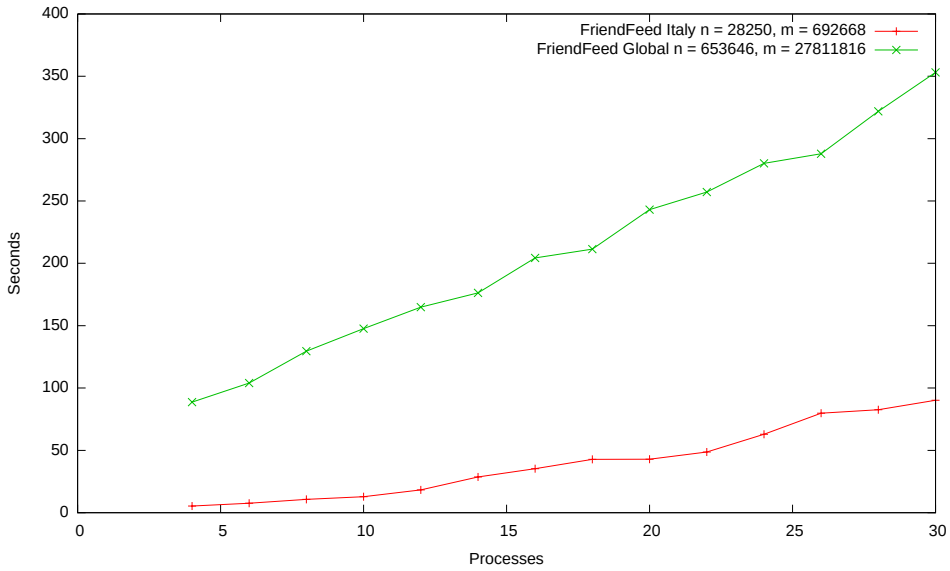


Figure 4.2: PageRank's wall clock time on FriendFeed's Global and Italian networks

the correlation coefficients between the wall clock time and the number of processors for all the experiments. Except for some bumps, also reflected in lower correlation coefficients, all the experiments showed a linear correlation between the wall clock time and the number of processors.

4.3.2 Impact of the graph type on scalability

Figure 4.5 compares the PageRank's wall clock time on various networks with 653646 nodes and about 27811816 edges. This graph shows how different graph types influence the scalability of the PageRank algorithm. Wall clock times are similar with a low number of processors, but as the number of processors increases, differences between graph types become more apparent. Graphs with an higher clustering coefficient are a better fit for the Parallel BGL's PageRank implementation. In fact, Small-World graphs "scale" better, in sharp contrast with Erdős–Rényi graphs that are the worst of the group. FriendFeed's Global network stays in between, having an higher clustering coefficient than Erdős–Rényi but lower than Small-World.

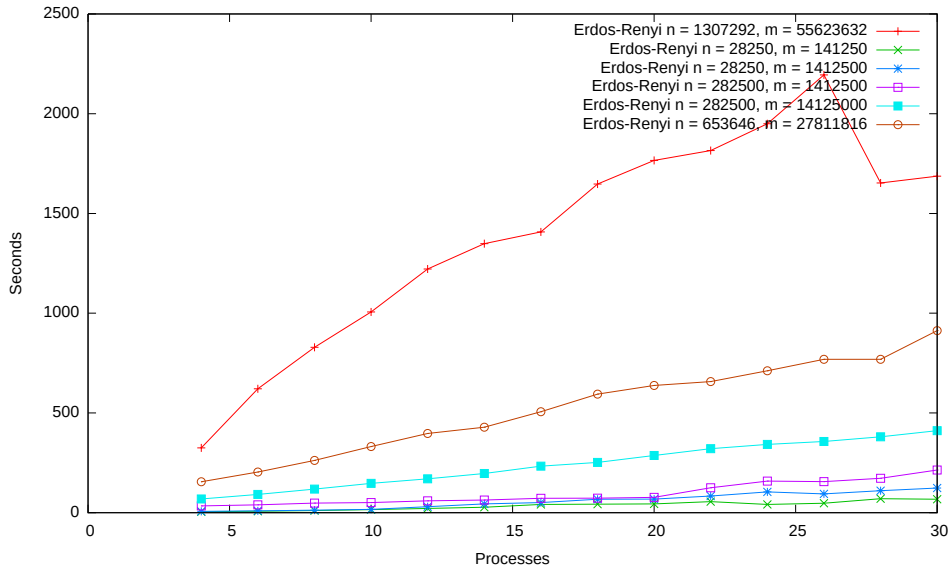


Figure 4.3: PageRank's wall clock time on Erdős–Rényi networks

The explanation of this phenomenon comes from the PageRank implementation. As previously mentioned in Section 3.2, the Parallel BGL's `page_rank_step` function updates the rank values of all the adjacent nodes for every node stored in the current process. Since the adjacent nodes can be stored either in the local or in a remote process, in graphs with an high clustering coefficient the communication time is greatly reduced. In graphs with a lower clustering coefficient processes need to communicate a lot more values between each other, thus increasing the wall clock time.

4.3.3 Impact of the edge count on scalability

Figure 4.6 highlights the relationship between wall clock times on graphs with different edge counts, taking Erdős–Rényi networks with 28250 nodes as an example. In this particular case, the wall clock time increased by 55.5376% on average, with a variance of 0.1332783, as the edge count increased tenfold, from 141250 to 1412500.

A similar experiment with ten times more nodes led to an even greater increase of the wall clock time. Figure 4.7 shows two experiments with

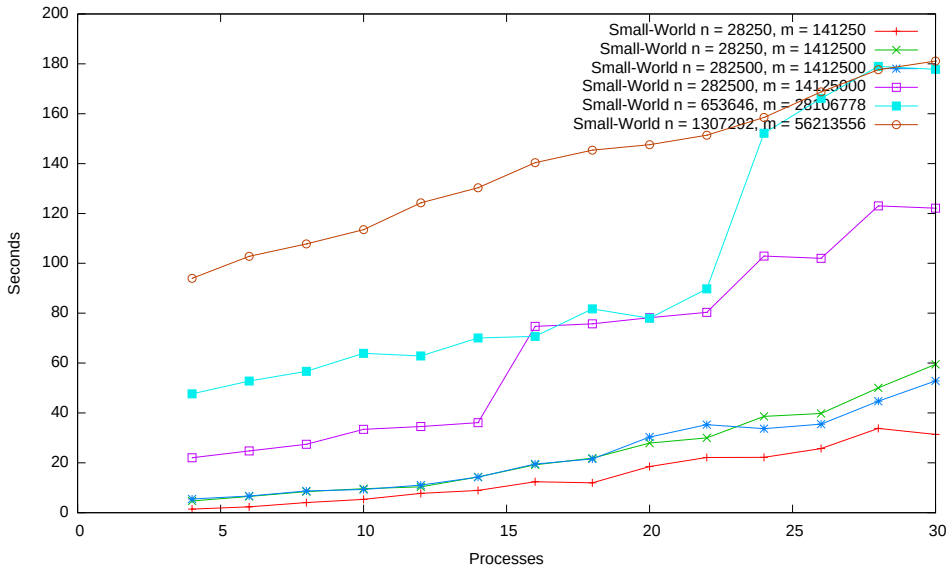


Figure 4.4: PageRank's wall clock time on Small-World networks

Erdős–Rényi networks of 282500 nodes and respectively, 1412500 and 14125000 edges. In this case, the wall clock time increased by 165.8727% on average, with a variance of 0.3132687.

Since a tenfold increase in the number of edges coincided with a 55.5376% increase of the wall clock time for networks of 28250 nodes, and with a 165.8727% increase for networks of 282500 nodes, we have the reason to suspect that the edge count is not directly related to the wall clock time but it plays a symbiotic role with the node count.

Graph Type	Nodes	Edges	$\rho_{p,t}$
FriendFeed Global	653646	27811816	0.9966577
FriendFeed Italian	28250	692668	0.9808243
Erdős–Rényi	28250	141250	0.9618445
Erdős–Rényi	28250	1412500	0.9883753
Erdős–Rényi	282500	1412500	0.9390956
Erdős–Rényi	282500	14125000	0.9986524
Erdős–Rényi	653646	27811816	0.9943021
Erdős–Rényi	1307292	55623632	0.900231
Small-World	28250	141250	0.9786595
Small-World	28250	1412500	0.9668417
Small-World	282500	1412500	0.9737682
Small-World	282500	14125000	0.9723917
Small-World	653646	28106778	0.9076243
Small-World	1307292	56213556	0.9957426

Table 4.2: Correlation coefficients between the number of processors p and the wall clock time t for all the experiments

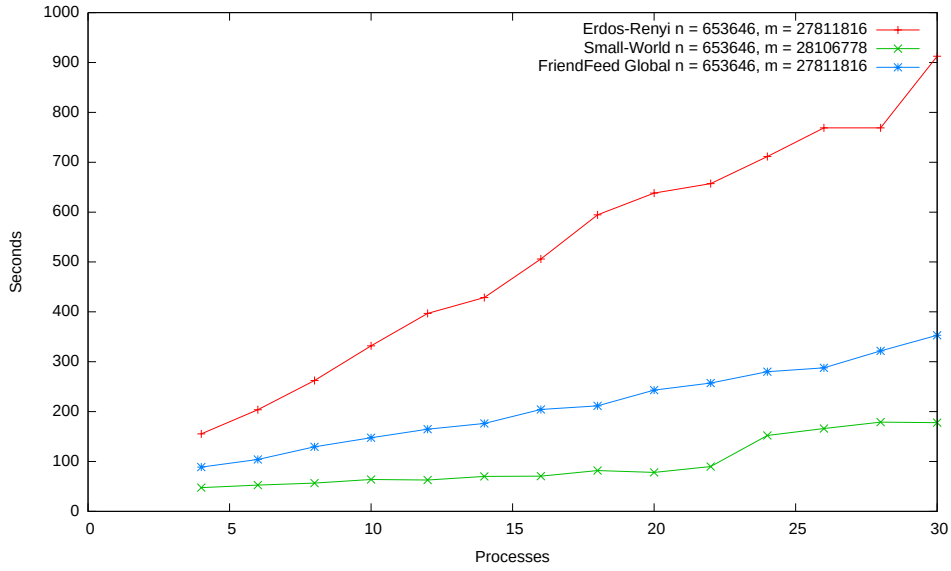


Figure 4.5: PageRank's wall clock time on networks with 653646 nodes

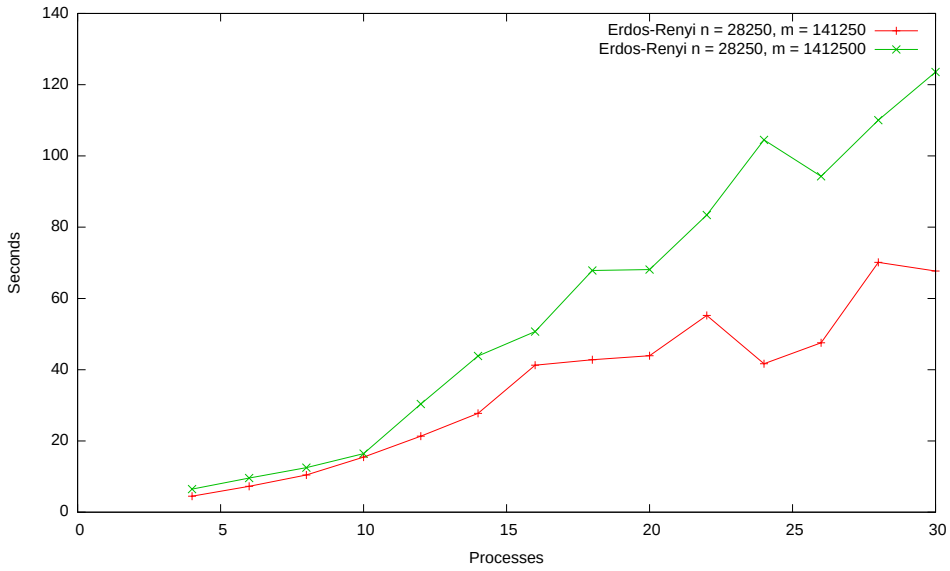


Figure 4.6: PageRank's wall clock time on Erdős–Rényi networks with 28250 nodes

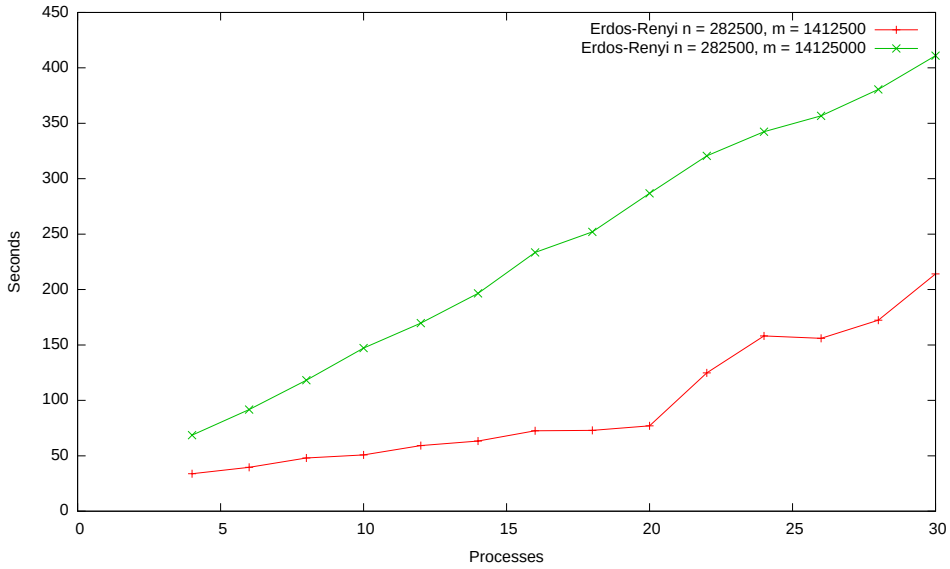


Figure 4.7: PageRank's wall clock time on Erdős–Rényi networks with 282500 nodes

Conclusions

This dissertation introduced graph theory and network analysis, overviewing the terminology, the centrality measures, and the graph models needed to better understand the PageRank implementation from the Parallel BGL. Then it presented the Parallel BGL, the fundamental data structures it provides, its consistency models, and its PageRank implementation. Finally, it analyzed the behavior of the PageRank implementation from the Parallel BGL in a cluster of commodity computers.

My tests showed that the Bulk Synchronous Parallel model of computing, used in the PageRank implementation from the Parallel BGL, is unsuitable to implement a PageRank algorithm that scales well on clusters of commodity computers. The PageRank implementation from the Parallel BGL exhibited, in fact, negative scalability. The wall clock time of the algorithm increased linearly as a function of the number of processors.

Experimental results showed a relationship between the clustering coefficient and the algorithm scalability. For example, networks with a low clustering coefficient, such as Erdős–Rényi ones, represented the worst case scenario of our tests, while networks with a high clustering coefficient, such as Small-World ones, represented the best case scenario. The difference in size of the networks also led to interesting results. In fact, tests revealed that the combined effect of node and edge count produces an increase of the wall clock time.

The pervasiveness and inexpensiveness of today's multi-core computers forces parallel programmers, historically trained to deal with supercomputers, to devote their attention to those new, different parallel environments. In the past few years a number of researchers developed fast parallel PageRank

implementations [GZB04] [ZYL05] [BdJKT05] [KCN06]. Further research is needed to implement those algorithms and to assess their performance in clusters of commodity computers.

Bibliography

[And00] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison Wesley, 2000.

[BdJKT05] Jeremy Bradley, Douglas de Jager, William Knottenbelt, and Aleksandar Trifunović. Hypergraph partitioning for faster parallel pagerank computation. In Mario Bravetti, Leïla Kloul, and Gianluigi Zavattaro, editors, *Formal Techniques for Computer Systems and Business Processes*, volume 3670 of *Lecture Notes in Computer Science*, pages 155–171. Springer Berlin / Heidelberg, 2005.

[Bor99] Shekhar Borkar. Design challenges of technology scaling. *Micro, IEEE*, 19(4):23 –29, 1999.

[BP98] Sergey Brin and Larry Page. The anatomy of a large-scale hypertextual web search engine. In *Seventh International World-Wide Web Conference (WWW 1998)*, 1998.

[CDLM+10] Fabio Celli, F. Di Lascio, Matteo Magnani, Barbara Pacelli, and Luca Rossi. Social network data and practices: The case of friendfeed. In Sun-Ki Chai, John Salerno, and Patricia Mabry, editors, *Advances in Social Computing*, volume 6007 of *Lecture Notes in Computer Science*, pages 346–353. Springer Berlin / Heidelberg, 2010.

[ER59] Paul Erdős and Alfréd Rényi. On random graphs I. *Publ. Math. Debrecen*, 6(290-297):156, 1959.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, 1972.

[Fre79] Linton C. Freeman. Centrality in social networks conceptual clarification. *Social Networks*, 1(3):215 – 239, 1978-1979.

[GL05] Douglas Gregor and Andrew Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[GZB04] David Gleich, Leonid Zhukov, and Pavel Berkhin. Fast Parallel PageRank: A Linear System Approach. 2004.

[KCN06] Christian Kohlschütter, Paul-Alexandru Chirita, and Wolfgang Nejdl. Efficient Parallel Computation of PageRank. In Mounia Lalmas, Andy MacFarlane, Stefan Rüger, Anastasios Tombros, Theodora Tsikrika, and Alexei Yavlinsky, editors, *Advances in Information Retrieval*, volume 3936 of *Lecture Notes in Computer Science*, pages 241–252. Springer Berlin / Heidelberg, 2006.

[OAS10] Tore Opsahl, Filip Agneessens, and John Skvoretz. Node centrality in weighted networks: Generalizing degree and shortest paths. *Social Networks*, 32(3):245 – 251, 2010.

[PBMW98] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford Digital Library Technologies Project, 1998.

[SLL02] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library*. Addison-Wesley Boston, MA, 2002.

[vNG93] John von Neumann and Michael D. Godfrey. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[WS98] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world'networks. *Nature*, 393(6684):440–442, 1998.

[ZYL05] Yangbo Zhu, Shaozhi Ye, and Xing Li. Distributed pagerank computation based on iterative aggregation-disaggregation methods. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, CIKM '05, pages 578–585, New York, NY, USA, 2005. ACM.

Acknowledgments

It is an honor for me to thank my supervisors, Moreno Marzolla and Matteo Magnani, whose encouragement, guidance and support from the first to the very last day enabled me to develop an understanding of the subject. I am also grateful to my parents, who backed me financially and emotionally.

Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of this project.