A middleware for parallel processing of large graphs

Tiago Alves Macambira and
Dorgival Olavo Guedes Neto
{tmacam,dorgival}@dcc.ufmg.br

National Institute for Web Research (InWeb)
DCC — UFMG — Brazil

MGC 2010 — 2010-11-30
Outline

1. Introduction
2. The API
3. Implementation
4. Evaluation
5. Conclusions
Introduction

From experimentation to “Data Deluge”

Collecting “large” datasets is dead simple(r) nowadays:

- We can easily and passively collect them electronically.
- Advances in data storage and computer processing made storing and processing such datasets feasible.

This has been beneficial to many different research fields such as:

- biology,
- computer science,
- sociology,
- physics,
- et cetera.
Introduction

“With great power comes great responsibility…”

On the other hand, extracting “knowledge” from such datasets has not been easy:

- Their sizes exceed what nowadays single node systems are capable of handling,
  - in terms of storage (be it primary or secondary storage)
  - in terms of processing power (considering a “reasonable” time)
- The use of distributed or parallel processing can mitigate such limitations.

If those datasets represent “relationship among entities” or a graph, the problem might just get worse.

- But what do we consider “huge” graphs?
- And why does it get worse?
## Huge graphs

Size of some graphs and their storage costs [Newman, 2003, Cha et al., 2010]:

<table>
<thead>
<tr>
<th>Description</th>
<th>n</th>
<th>m</th>
<th>$n^2$ (TiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electronic Circuits</td>
<td>24,097</td>
<td>53,248</td>
<td>0.002</td>
</tr>
<tr>
<td>Co-authorship (Biology)</td>
<td>1,502,251</td>
<td>11,803,064</td>
<td>8</td>
</tr>
<tr>
<td>LastFM (social network view)</td>
<td>3,096,094</td>
<td>17,220,985</td>
<td>34</td>
</tr>
<tr>
<td>Phone Calls</td>
<td>47,000,000</td>
<td>80,000,000</td>
<td>8,036</td>
</tr>
<tr>
<td>Twitter</td>
<td>54,981,152</td>
<td>1,963,263,821</td>
<td>10,997</td>
</tr>
<tr>
<td>WWW (Altavista)</td>
<td>203,549,046</td>
<td>2,130,000,000</td>
<td>150,729</td>
</tr>
</tbody>
</table>

Note: Storage needs considering a 32-bit architecture.
Parallel processing

“Freelunch is over” [Sutter, 2005]

- CPU industry struggled to keep the GHz race going.
- Instead of increasing clock speed, increase the number of cores.
- Multi-core CPU are not the exception but the rule.

{Parallel, distributed, cloud} computing is mainstream now, right?

- Yet, programmers still think it is hard — thus error-prone.
- There still is a need for better/newer/easier/more reliable:
  - abstractions,
  - languages,
  - frameworks,
  - paradigms,
  - models,
  - you name it
Parallel processing of (huge) graphs

Graph algorithms are notoriously difficult to parallelize.

- Algorithms have high
  - computational complexity and
  - storage complexity.
- Challenges for efficient parallelism [Lumsdaine et al., 2007]:
  - Data-driven computation.
  - Data is irregular.
  - Poor locality.
  - High access-to-computation ratio.
Related work

Approaches for (distributed) graph processing

Shared-memory systems (SMP) [Madduri et al., 2007]

- Graphs are way too big to fit into main and even secondary memory.
- Systems such as Cray MTA-2 are not viable from an economical standpoint.
Related work

Approaches for (distributed) graph processing

Distributed Memory Systems

- Message Passing
  - Writing application is considerably hard — thus, error prone.
- MapReduce
  - Graph Twiddling... [Cohen, 2009]
  - PEGASUS [Kang et al., 2009]
- Bulk Synchronous Parallel (BSP)
  - Pregel [Malewicz et al., 2010]
- Filter-Stream
  - MSSG [Hartley et al., 2006]
Goals

We think that a proper solution for this problem should:

- be usable on today’s clusters or cloud computing facilities
- be able to distribute the cost of storing and executing an algorithm in a large graph
- provide a convenient and easy abstraction for defining a graph processing application
Rendero

Is BSP-based model and uses a Vertex-oriented paradigm.

- Execution progresses in stages or *supersteps*.
- Each vertex in the graph is seen as a virtual processing unit.
  - Think “co-routines” instead of “threads”.
- During each *superstep*, each vertex (or node) can execute, conceptually in parallel, a user provided function,
- Messages sent during the course of a *superstep* are only delivered at the start of the next *superstep*. 
Rendero

During each *superstep*, each vertex (or node) can

- perform, conceptually in parallel, a user provided function,
- in which it can …
  - send messages (to other vertices),
  - process received messages,
  - “vote” for the execution of the next *superstep*,
  - “output” some result.

An execution terminates when all nodes abstain from voting.

From a programmer’s perspective, writing a Rendero program translates into defining two C++ classes:

- **Application**, that deals with resource initialization and configuration before an execution begins.
- **Node**.
Nodes

Define what each vertex in graph must do during each superstep by means of 3 user-defined functions:

- `onStep()` — what must be done on each superstep.
- `onBegin()` — what must be done on the first superstep.
- `onEnd()` — what must be done after the last superstep.

Nodes have limited knowledge of the graph topology. Upon start each node only knows:

- its own identifier and
- its direct neighbors’ identifiers.

Nodes lack communication and I/O primitives, and rely on its Environment for those.
Environment

An abstract entity that provides communication and I/O primitives for nodes to:

- send messages: `sendMessage()`
- manifest their intent (or vote) on continuing the program’s execution: `voteForNextStep()`
- output any final or intermediary result: `outputResult()`

Messages and any output result are seen as untyped byte arrays.

- If needed, object serialization solutions such as Google Protocol Buffers, ApacheThrift or Avro can be employed.
Implementation

Rendero is coded in C++.

Allows for two forms of execution of the same user-provided source code:

- Sequential
  - Handy for tests and debugging on small graphs.
- Distributed
  - For processing large graphs
Components

Nodes

- Used-defined by subclassing the BaseNode.

Node Containers

- A storage and management facility for Node instances.
- Provide a concrete implementation of an Environment for their nodes.
- Implement message routing and sorting logic.
  - In a distributed execution, nodes are currently assigned to Containers using a simple hash function on their identifiers.

Conductor

- coordinates an (distributed) execution,
- orchestrates Containers actions,
- aggregates and broadcasts “election” results.
Out-of-core message storage

Problem:

- The number of messages issued during a superstep can exceed a system’s memory.
- OTOH, messages must be stored until the start of the following superstep.
  - There is no speculative execution.
  - All messages targeted to a given node must be delivered to it at once, during the invocation of its onStep() method.

Solution: Store these messages out-of-core.

- Containers periodically flush received messages to disk in blocks or runs.
- At the beginning of the following superstep, a multi-way merge of the runs is performed.
- The amount of primary memory is kept under control.
Example application: Connected Components

Description

Goal:

- find out all Connected Components of a graph.

Intuitively:

- We will run a distributed “election” to find out which node, in a given component, has the smallest identifier — that is going to be our “component head”.
- Upon start, each vertex starts a *flooding* of its identifier;
- During each *superstep*, each node forwards for its neighbors only the smallest identifiers it finds out.
- An execution is over on the *superstep* in which no node discovered new and smaller identifiers.
void onBegin(const mailbox_t& inbox) {
    // my_component_ is an instance variable
    my_component_ = this->getId();
    // broadcast my current component ID to my neighbours
    sendScalarToNeighbours(my_component_);
    // voting in the 1st step is optional,
    // but let’s do it anyway
    env_->voteForNextStep();
}
void onStep(const mailbox_t& inbox) {
    //typedef uint64_t cid_t;
    cid_t old_component_id = my_component_;  

    mailbox_t::iterator mi;
    for (mi = inbox.begin(); mi != inbox.end(); ++mi) {
        const cid_t& nei_component = MsgToScalar(*mi);
        my_component_ = std::min(my_component_,
                                 nei_component);
    }
    if (old_component_id > my_component_) {
        // Better component head found.
        // Notify neighbours
        sendScalarToNeighbours(my_component_);
        env_->voteForNextStep();
    }
}
Connected Components

```c++
void onEnd() {
    std::ostringstream os;
    os << "Id " << this->getId() << " comp " << my_component_; 
    env_->outputResult(os.str());
}
```
Other applications

Other applications were developed:

- **Single-source shortest path**
  - Finds the shortest distance from a root node to all other nodes in a graph.

- **All-pairs shortest path**
  - Finds out the distance between any two nodes in a graph.
  - As seen in the introduction, has strong storage requirements.

- Clustering Coefficient.

- PageRank
Experimental Evaluation

Environment description:

- A 13-node dual-core cluster
- Intel Core2 6420 processors
- 2 GB RAM
- Linux, using kernel 2.6.22-14-server from Ubuntu.

Algorithms:

- Connected components
- All-pairs shortest path
Datasets description

<table>
<thead>
<tr>
<th>Base</th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wikipedia</td>
<td>7,115</td>
<td>103,689</td>
</tr>
<tr>
<td>Cond. Matter</td>
<td>23,133</td>
<td>186,936</td>
</tr>
<tr>
<td>Slashdot09</td>
<td>82,168</td>
<td>948,464</td>
</tr>
<tr>
<td>LastFM</td>
<td>3,096,094</td>
<td>17,220,985</td>
</tr>
</tbody>
</table>

Table: Major features of the datasets used

Datasets from: [Leskovec, 2010]
Results

*Speed-up for Connected Components application*

![Graph showing speed-up for components application across different datasets and number of nodes. The graph includes linear speed-up lines for LastFM and Slashdot datasets.](graph.png)
Resultados

Scaled speed-up for All-pairs shortest path application

Scaled speed-up for APSP application

Linear Speed-up
Condensed matter dataset
Wikipedia votes dataset
Conclusions

The middleware is implemented and reaches its goals:

- it is easy to use and
- handles large graphs in modest-sized clusters.

Its out-of-core facilities are transparent to the end-user

- Yes, they do incur in run-time penalties,
- But is a fair trade-off, given our objectives.

We noticed speed-up gains for applications written to it

- But we need to enhance its monitoring capabilities to better understand its I/O and communication load.
Next steps and future work

- Properly open-sourcing this code.
- Study graph partitioning and load balance strategies.
- Support application chaining, graph and meta-data persistence.
- Experiment with speculative asynchronous execution strategies and how they can be made without modifying the user-facing API and model.
Bibliography I


Bibliography II


The structure and function of complex networks.  

The free lunch is over: A fundamental turn toward concurrency in software.  
Thank you

Obrigado.
Questions?

Doubts? Suggestions?

Contact us

{tmacam, dorgival}@dcc.ufmg.br