THE LANDSCAPE OF LARGE SCALE GRAPH PROCESSING:
A VIEW FROM HOLLAND

Ana Lucia Varbanescu, University of Amsterdam, The Netherlands
a.l.varbanescu@uva.nl

With data, work, and (some) slides from a whole team:
Merijn Verstraaten, Ate Penders, Yong Guo, Alexandru Iosup, and others.

What to do when your graphs get out of control?
In this talk …

- Graph Analytics = any form of graph processing
- Platform = hardware and/or software we can tune and change as a whole
- (Graph) Processing system = computing system that includes one or more platforms (for graph processing)
Today’s headlines

1. Graphs and graph processing
2. Benchmarking I: Algorithms
3. Benchmarking II: Platforms
4. Future research directions
5. Take home message
Graphs and graph processing
Graph analytics at work
Numbers ...
In April 2014 ...

We now have 300 million LinkedIn members, more than half of whom live outside of the U.S. That’s enough to make LinkedIn the fourth largest country in the world. In celebration, we took a look back to see how much our membership has grown and diversified over the past five years. It’s a helpful reminder of not only where we’ve been, but also where we’re headed as we work to create economic opportunity for every professional in the world.
Classical analytics

- **Statistics**
  - “How many connections do I have?”

- **Traversing**
  - “How can I reach Prof. X?”

- **Querying**
  - “Find all professionals in Graph Processing around Dresden.”

- **Mining**
  - “Find the most influential CS researcher in Amsterdam.”
Classical analytics

- **Statistics**
  - “How many connections do I have?”

- **Traversing**
  - “How can I reach Prof. X?”

- **Mining**
  - “Find the most influential CS researcher in Amsterdam.”

No textbook algorithms exist for some of these operations. If they exist, they probably need changing.
Your network is so large...

Sorry, but your network is too large to be computed, we are working to increase the limit, stay tuned!
Large Scale, Graph Processing

- **Large-scale**
  - Very large data
    - Partitioning and parallel processing are mandatory!
  - Complex analytics
    - Absolute or approximate …
  - Data might evolve in time
    - Fast processing or new algorithms?

- **Graph processing**
  - Data-driven computations
  - Irregular memory accesses
    - Poor data locality
  - Unstructured problems
  - Low computation-to-data access ratio
Large Scale Graph Processing

- Graph processing is (very) data-intensive
  - 10x larger graph => 100x or 1000x slower processing
- Graph processing becomes (more) compute-intensive
  - More complex queries => ?x slower processing
- Graph processing is (very) dataset-dependent
  - Unfriendly graphs => ?x slower processing

High performance enables larger graphs and support for more complex analytics.
More performance? Many-cores!
Top500 in November 2014

- Traditional HPC is about computing ... not graphs!

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>System</th>
<th>Cores</th>
<th>Rmax (TFlop/s)</th>
<th>Rpeak (TFlop/s)</th>
<th>Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National University of Defense Technology, China</td>
<td>Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.20GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT</td>
<td>3120000</td>
<td>33862.7</td>
<td>54902.4</td>
<td>17808</td>
</tr>
<tr>
<td>2</td>
<td>DOE/SC/Oak Ridge National Laboratory, United States</td>
<td>Titan - Cray XK7, Opteron 6274 16C 2.20GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.</td>
<td>560640</td>
<td>17590.0</td>
<td>27112.5</td>
<td>8209</td>
</tr>
<tr>
<td>3</td>
<td>DOE/NNSA/LLNL, United States</td>
<td>Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM</td>
<td>1572864</td>
<td>17173.2</td>
<td>20132.7</td>
<td>7890</td>
</tr>
<tr>
<td>4</td>
<td>RIKEN Advanced Institute for Computational Science (AICS), Japan</td>
<td>K computer, SPARC64 Vlllfx 2.0GHz, Tofu interconnect Fujitsu</td>
<td>705024</td>
<td>10510.0</td>
<td>11280.4</td>
<td>12660</td>
</tr>
<tr>
<td>5</td>
<td>DOE/SC/Argonne National Laboratory, United States</td>
<td>Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM</td>
<td>786432</td>
<td>8586.6</td>
<td>10066.3</td>
<td>3945</td>
</tr>
<tr>
<td>Rank</td>
<td>Machine</td>
<td>Installation Site</td>
<td>Number of nodes</td>
<td>Number of cores</td>
<td>Problem scale</td>
<td>GTEPS</td>
</tr>
<tr>
<td>------</td>
<td>--------------------------------------------------------------------------</td>
<td>------------------------------------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>---------------</td>
<td>--------</td>
</tr>
<tr>
<td>1</td>
<td>K computer (Fujitsu - Custom supercomputer)</td>
<td>RIKEN, Japan</td>
<td>65536</td>
<td>524288</td>
<td>40</td>
<td>17977</td>
</tr>
<tr>
<td>2</td>
<td>DOE/NNSA/LLNL Sequoia (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz)</td>
<td>LLNL, USA</td>
<td>65536</td>
<td>1048576</td>
<td>40</td>
<td>16599</td>
</tr>
<tr>
<td>3</td>
<td>DOE/SC/Argonne National Laboratory Mira (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz)</td>
<td>ANL, USA</td>
<td>49152</td>
<td>786432</td>
<td>40</td>
<td>14328</td>
</tr>
<tr>
<td>4</td>
<td>JUQUEEN (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz)</td>
<td>FZJ, Germany</td>
<td>16384</td>
<td>262144</td>
<td>38</td>
<td>5848</td>
</tr>
<tr>
<td>5</td>
<td>Fermi (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz)</td>
<td>CINECA</td>
<td>8192</td>
<td>131072</td>
<td>37</td>
<td>2567</td>
</tr>
<tr>
<td>6</td>
<td>Tianhe-2 (MilkyWay-2) (National University of Defense Technology - MPP)</td>
<td>Changsha, China</td>
<td>8192</td>
<td>196608</td>
<td>36</td>
<td>2061</td>
</tr>
<tr>
<td>7</td>
<td>Turing (IBM - BlueGene/Q, Power BQC 16C 1.60GHz)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Blue Joule (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz)</td>
<td>Laboratory, UK</td>
<td>4096</td>
<td>65536</td>
<td>36</td>
<td>1427</td>
</tr>
<tr>
<td>9</td>
<td>DIRAC (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz)</td>
<td>University of Edinburgh, UK</td>
<td>4096</td>
<td>65536</td>
<td>36</td>
<td>1427</td>
</tr>
<tr>
<td>10</td>
<td>Zumbrota (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz)</td>
<td>EDF R&amp;D</td>
<td>4096</td>
<td>65536</td>
<td>36</td>
<td>1427</td>
</tr>
</tbody>
</table>
The challenges

- **Feasibility:**
  Can we use multi-core and many-core processors – to address the performance requirements for modern graph algorithms?

- **Usability:**
  Is there a systematic solution to enable this match?
Why challenging?

- Many-cores have emerged to improve performance by using massive parallelism.

- Performance gain in theory: $N$ cores $\Rightarrow N$ times faster

- For this, we need:
  - massive (multi-layered) parallelism
  - high computation-to-data access ratio
  - high data locality
  - structured, regular access patterns

Remember graph processing?
- Data-driven computations
- Irregular memory accesses
  - Poor data locality
- Unstructured problems
  - Low computation-to-data access ratio
Additional challenge

Which one to choose?!?!
Can we run graph analytics on HPC architectures, efficiently?
BFS ➔ APSP ➔ BC

- **Graph traversal (Breadth First Search, BFS)**
  - Traverses all vertices “in levels”

- **All-Pairs Shortest Paths (APSP)**
  - Repeat BFS for each vertex

- **Betweenness Centrality (BC)**
  - APSP once to determine paths
  - Bottom-up BFS to count paths

- **Implementation in OpenCL**
  - Same algorithm
  - CPU- and GPU-specific **tuning** applied

*Ate Penders MSc thesis
“Accelerating graph processing using modern accelerators”*
## Data sets & devices

<table>
<thead>
<tr>
<th>Data set</th>
<th>Abbreviation</th>
<th>Vertices</th>
<th>Edges</th>
<th>Diameter</th>
<th>Avg. Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wikipedia Talk Network</td>
<td>WT</td>
<td>2,394,385</td>
<td>5,021,410</td>
<td>9</td>
<td>2.10</td>
</tr>
<tr>
<td>California Road Network</td>
<td>CR</td>
<td>1,965,206</td>
<td>5,533,214</td>
<td>850</td>
<td>2.81</td>
</tr>
<tr>
<td>Rodinia Graph 1M</td>
<td>1M</td>
<td>1,000,000</td>
<td>6,000,000</td>
<td>36</td>
<td>6.00</td>
</tr>
<tr>
<td>Stanford Web Graph</td>
<td>SW</td>
<td>281,903</td>
<td>2,312,497</td>
<td>740</td>
<td>8.20</td>
</tr>
<tr>
<td>EU Email Communication Network</td>
<td>EU</td>
<td>265,214</td>
<td>420,045</td>
<td>13</td>
<td>1.58</td>
</tr>
<tr>
<td>Star</td>
<td>ST</td>
<td>100,000</td>
<td>99,999</td>
<td>1</td>
<td>0.99</td>
</tr>
<tr>
<td>Chain</td>
<td>CH</td>
<td>100,000</td>
<td>99,999</td>
<td>99,999</td>
<td>1.00</td>
</tr>
<tr>
<td>Epinions Social Network</td>
<td>ES</td>
<td>75,879</td>
<td>508,837</td>
<td>13</td>
<td>6.70</td>
</tr>
<tr>
<td>Rodinia Graph 64K</td>
<td>64K</td>
<td>64,000</td>
<td>393,216</td>
<td>28</td>
<td>6.14</td>
</tr>
<tr>
<td>Wikipedia Vote Network</td>
<td>VW</td>
<td>7,115</td>
<td>103,689</td>
<td>7</td>
<td>14.57</td>
</tr>
<tr>
<td>Rodinia Graph 4K</td>
<td>4K</td>
<td>4000</td>
<td>25,356</td>
<td>19</td>
<td>6.38</td>
</tr>
</tbody>
</table>

### Devices

- Intel(R) Xeon(R) CPU E5620 @ 2.40GHz
- GeForce GTX 480
- Tesla C2050 / C2070
BFS – normalized

[Bar chart comparing normalized BFS for different configurations using Xeon (CPU), Tesla (GPU), and GTX (GPU).]
BFS – normalized

Performance depends on the diameter and degree:
Large diameter => CPU
High degree => GPU
APSP - normalized

- Xeon (CPU)
- Tesla (GPU)
- GTX (GPU)
GPUs always win due to the (enforced) high parallelism of our solution.
BC - normalized

![Bar chart showing normalized BC results for different categories such as WT, CR, 1M, SW, EU, CH, ST, ES, 64K, WV, and 4K. The categories are represented by different colored bars: red for Xeon (CPU), green for Tesla (GPU), and blue for GTX (GPU).](chart.png)
Atomic operations for counting paths => variable performance due to variable contention!
Graphs seem to be CPU or GPU friendly

- Data-dependent performance variations using the same implementation
  - CPU = lower parallelism, more caching
  - GPU = massive parallelism, less caching

- Memory size is an issue!
  => true large scale?
Lessons learned [2]

- Increased algorithm complexity can increase parallelism
  - E.g.: ASPS = $|V| \times$ BFS

- Dataset representation and properties increase parallelism

- Synchronization is an important bottleneck
  - E.g.: BC mixes compute with synchronization

- We have no clear understanding of graph “sizes”
  - # vertices or # edges?
  - Diameter?
  - Other properties?
Experiment 2: BFS traversals

- **Question:**
  - Is there a best BFS algorithm?
    - On GPUs?
    - Overall?

- **Setup:**
  - Run multiple BFS implementations
    - Including the ones claimed to be the best @ LonestarGPU
  - Run on different graphs
    - 6 datasets
  - Run on different hardware
Normalized on naïve GPU, kernel
Orders of magnitude performance difference. No clear winner.
Normalized on naïve GPU, full exec.
Adding the data transfer times narrows the performance gaps.
Lessons learned [1]

- Large variability in performance depending on the graph.
  - Fastest to slowest ratio varies.

- The relative performance of one BFS implementation varies for different graphs.
  - Fastest on one graph CAN BE slowest on another graph.

- Data representation and data structures make the difference
  - List of edges vs adjacency lists
  - Lock-free frontier
Lessons learned [2]

- Two disjoined classes of algorithms
  - GPU-optimized
  - Portable to CPU (= naïve)
- A naive CPU implementation can be competitive with some of the GPU implementations.
  - On small graphs (GPUs are underutilized)
  - When data transfer is an issue.
- Better CPU solutions do exist ...
Summary
Take home message

- Large scale graph processing IS high performance computing
  - Due to/for data scale *and* analysis complexity
- HPC hardware (many-core processors) are feasible for graph processing
  - yet performance is (for now) unpredictable
- Performance is dependent on all three “axes”
  - Performance = f (dataset, algorithm, hardware)
P-A-D triangle

Algorithm

In progress
 Algorithms for different data types and graphs

Overstudied
 Performance is enabled
 Portability is disabled

Dataset

Understudied
 No systematic findings yet
 Intuitive correlations
 Must be correlated with the algorithm

Platform
Benchmarking II: Platforms
Graph processing @ scale

- The characteristics of graph processing
  - Poor locality
  - Unstructured computation
  - Variable parallelism
  - Low computer-to-memory ratio

- @ Scale
  - Distributed processing is mandatory
  - Parallel processing is very useful

Implementing graph applications is already difficult. Dealing with large scale systems on top (below, in fact) them is even harder.
Graph processing systems

- Provide simplified ways to develop graph processing applications
  - Typical scenario: analytics on single- or multi-node platforms
  - Heterogeneity is becoming popular

- Target *productivity* and *performance*
  - Productivity => ease-of-implementation, development time
  - Performance => optimized back-ends / engines / runtimes
  - Portability comes “for free”

- Both commercial and academic, many open-source
Graph processing systems

- Systems for graph processing
- Separate users from backends
- Think Giraph, Totem, Medusa, ....

- Use existing large scale distributed systems
- Mapping is difficult
- Parallelism is “free”
- Think MapReduce

Custom

- Specify application
- Choose the hardware
- Implement & optimize
- Think Graph500

Dedicated Systems

Generic

Performance

Development Effort
GPU-enabled dedicated systems
MapGraph*

- GPU-only graph processing
  - CPU, single- and multi-GPU versions
- Vertex-centric API based on Gather-Apply-Scatter (GAS @ GraphLab)
  - Gather: reads the vertex’s neighborhood.
  - Apply: updates the vertex based on the gather result.
  - Scatter: pushes updates to the vertex’s neighborhood.
    - Users write functions for the G-A-S phases
- Two data structures (user-defined)
  - VertexList: data for each vertex.
  - EdgeList: data for each edge

*http://mapgraph.io/index.html
Medusa*

- GPU-only graph processing
  - Single-node, multiple GPUs
- Programmability-driven, based on BSP
  - EMV (edge-message-vertex) model
    - Extension of the Vertex-centric Pregel-like model
  - GPU-specific back-end optimization
- Simple API that hides GPU programming
  - Define data structures
  - Define operations for edges, messages, vertices
  - Compose the algorithm from these operations
  - Run (iteratively) over the graph

*https://code.google.com/p/medusa-gpu/*
Totem*

- Heterogeneous CPU+GPUs graph processing
  - C+CUDA for specifying applications
  - (Thin) API for heterogeneity
  - Based on BSP (and close to Pregel)
- Partitions data (edge-based) between CPUs and GPUs
  - Based on processing capacity
  - Minimizing the overhead of communication
    - Buffer schemes, aggregation, smart partitioning
- A user-defined vertex-centric kernel runs simultaneously on each partition (CPUs, GPUs)
  - Vertices are processed in parallel within each partition
  - Messages can be combined

*http://netsyslab.ece.ubc.ca/wiki/index.php/Totem
Experiments*

Setup: Algorithms & Systems

- **Algorithms**
  - BFS (traversal)
  - PageRank
  - Weakly connected components

- **Hardware: GPU-enabled nodes in DAS4**
  - GTX480 (most results), GTX580, and K20

- **Processing systems:**
  - Totem - GPU-only and Hybrid
  - Medusa – single- and multi-GPU
  - MapGraph – single-GPU
### Setup: datasets

<table>
<thead>
<tr>
<th>Graphs</th>
<th>$V$</th>
<th>$E$</th>
<th>$d$</th>
<th>$\bar{D}$</th>
<th>Max $D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon (D)</td>
<td>262,111</td>
<td>1,234,877</td>
<td>1.8</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>WikiTalk (D)</td>
<td>2,388,953</td>
<td>5,018,445</td>
<td>0.1</td>
<td>2</td>
<td>100,022</td>
</tr>
<tr>
<td>Citation (D)</td>
<td>3,764,117</td>
<td>16,511,742</td>
<td>0.1</td>
<td>4</td>
<td>770</td>
</tr>
<tr>
<td>KGS (U)</td>
<td>293,290</td>
<td>22,390,820</td>
<td>26.0</td>
<td>76</td>
<td>18,969</td>
</tr>
<tr>
<td>DotaLeague (U)</td>
<td>61,171</td>
<td>101,740,632</td>
<td>2,719.0</td>
<td>1,663</td>
<td>17,004</td>
</tr>
<tr>
<td>Scale-22 (U)</td>
<td>2,394,536</td>
<td>128,304,030</td>
<td>2.2</td>
<td>54</td>
<td>163,499</td>
</tr>
<tr>
<td>Scale-23 (U)</td>
<td>4,611,439</td>
<td>258,672,163</td>
<td>1.2</td>
<td>56</td>
<td>257,910</td>
</tr>
<tr>
<td>Scale-24 (U)</td>
<td>8,870,942</td>
<td>520,760,132</td>
<td>0.7</td>
<td>59</td>
<td>406,417</td>
</tr>
<tr>
<td>Scale-25 (U)</td>
<td>17,062,472</td>
<td>1,047,207,019</td>
<td>0.4</td>
<td>61</td>
<td>639,144</td>
</tr>
</tbody>
</table>

$V$ and $E$ are the vertex count and edge count of the graphs. $d$ is the link density ($\times 10^{-5}$). $\bar{D}$ is the average vertex out-degree. Max $D$ is the largest out-degree. (D) and (U) stands for the original directivity of the graph. For each original undirected graph, we transfer it to directed graph (see Section II-B1).
## Setup: datasets

<table>
<thead>
<tr>
<th>Graphs</th>
<th>V</th>
<th>E</th>
<th>d</th>
<th>$\bar{D}$</th>
<th>Max $D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon (D)</td>
<td>262,111</td>
<td>1,234,877</td>
<td>1.8</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>WikiTalk (D)</td>
<td>2,388,953</td>
<td>5,018,445</td>
<td>0.1</td>
<td>2</td>
<td>100,022</td>
</tr>
<tr>
<td>Citation (D)</td>
<td>3,764,117</td>
<td>16,511,742</td>
<td>0.1</td>
<td>4</td>
<td>770</td>
</tr>
<tr>
<td>KGS (U)</td>
<td>293,290</td>
<td>22,390,820</td>
<td>26.0</td>
<td>76</td>
<td>18,969</td>
</tr>
<tr>
<td>DotaLeague (U)</td>
<td>61,171</td>
<td>101,740,632</td>
<td>2,719.0</td>
<td>1,663</td>
<td>17,004</td>
</tr>
<tr>
<td>Scale-22 (U)</td>
<td>2,394,536</td>
<td>128,304,030</td>
<td>2.2</td>
<td>54</td>
<td>163,499</td>
</tr>
<tr>
<td>Scale-23 (U)</td>
<td>4,611,420</td>
<td>258,672,163</td>
<td>1.2</td>
<td>56</td>
<td>257,016</td>
</tr>
<tr>
<td>Scale-25 (U)</td>
<td>17,062,472</td>
<td>1,047,207,019</td>
<td>0.4</td>
<td>61</td>
<td>639,144</td>
</tr>
</tbody>
</table>

$V$ and $E$ are the vertex count and edge count of the graphs. $d$ is the link density ($\times 10^{-5}$). $\bar{D}$ is the average vertex out-degree. Max $D$ is the largest out-degree. (D) and (U) stands for the original directivity of the graph. For each original undirected graph, we transfer it to directed graph (see Section II-B1).
BFS [algorithm]

Strong dependency on the graph.
Totem is the worst performer.
Medusa and MapGraph cannot handle large graphs.
BFS [full]

Totem becomes the best performer!
WCC [algorithm]

Strong dependency on the graph.
No best/worst performer.
More crashes of MapGraph.
PageRank [algorithm]

Most compute-intensive. Totem performs worst.
For large graphs, Totem-GPU is worse than hybrid.
Multi-GPU scalability

Platforms can use multiple GPUs efficiently. Load balancing matters.
GPU versions

No guaranteed gain for newer GPUs
Larger graphs seem to benefit more from K20m.
Lessons learned

- Brave attempts to enable the use of GPUs *inside* graph processing systems
- Every system has its own quirks
  - Lower level programming allows more optimizations, better performance.
  - Higher level APIs allow more productivity.
- Data pre-processing and data structure are crucial to both performance and capability.
- No clear winner, performance-wise.
Distributed/Large Scale platforms
Graph processing systems

- Systems for graph processing
- Separate users from backends
- Think Giraph, Totem, Medusa, ....

Dedicated Systems

- Use existing large scale distributed systems
- Mapping is difficult
- Parallelism is “free”
- Think MapReduce

Generic

- Specify application
- Choose the hardware
- Implement & optimize
- Think Graph500

Custom

Performance

Development Effort
Hadoop (Generic)

- The most popular MapReduce implementation
  - Generic system for large-scale computation

- Pros:
  - Easy to understand model
  - Multitude of tools and storage systems

- Cons:
  - Express the graph application in the form of MapReduce
  - Costly disk and network operations
  - No specific graph processing optimizations
Hadoop2 with YARN (Generic)

- Next generation of Hadoop
  - Supports old MapReduce jobs
  - Designed to facilitate multiple programming models (frameworks, e.g., Spark)
- Separates resource management (YARN) and job management
  - MapReduce manages jobs using resources provided by YARN
Stratosphere (Generic)

- Now Apache Flink
- Nephele resource manager
  - Scalable parallel engine
  - Jobs are represented as DAGs
  - Supports data flow in-memory, via network, or on files
- PACT job model
  - 5 second-order functions (MapReduce has 2): Map, Reduce, Match, Cross, and CogGroup
  - Code annotations for compile-time plans
  - Compiled as DAGs for Nephele
Pregel graph-processing model

- Proposed a **vertex-centric** approach to graph processing
  - Graph-to-graph transformations

- Front-end:
  - Write the computation that runs on all vertices
  - Each vertex can vote to halt
    - All vertexes halt => terminate
  - Can add/remove edges and vertices

- Back-end:
  - Uses the BSP model
  - Message passing between nodes
    - Combiners, aggregators
  - Checkpointing for fault-tolerance
Pregel

Processing Model:
All “active” node will be executed
Whole processing completed when
a. No more active node
b. No more in-transit messages

Superstep execution:
1. Receive message from inbox
2. Modify node and arc properties
3. Halt self (until new message received)
4. Send messages to other nodes (causing them active)
5. Remove existing or create new arcs
Pregel

Aggregate value using Tree-based reduction

Master

Lock-step synchronization

Worker

InQ → Worker
outQ

Load and Checkpoint

Worker

InQ → Worker
outQ

Graph DB

Messages between 2 nodes can be combined
Apache Giraph (Dedicated)

- Based on the Pregel model
- Uses YARN as back-end (yet another framework)
- In-memory
  - Limitations in terms of partition sizes
  - Spilling to disk is work in progress
- Enables
  - Iterative data processing
  - Message passing, aggregators, combiners
GraphLab (Dedicated)

- Distributed programming model for machine learning
  - Provides an API for graph processing, C++ based (now Python)

- All in-memory
- Supports asynchronous processing
- GraphChi is its single-node version, Dato as GraphLab company
**Neo4J (Dedicated)**

- Very popular graph database
  - Graphs are represented as relationships and annotated vertices
- Single-node system
  - Uses parallel processing
  - Additional caching and query optimizations
  - All in-memory
- The most widely used solutions for medium-scale problems
Experiments*

Platforms we have evaluated

- Distributed or non-distributed
- Dedicated or generic

- Hadoop (Distributed (Generic))
- Giraph (Distributed (Dedicated))
- Neo4j (Non-distributed (Dedicated))
- YARN (Distributed (Generic))
- GraphLab (Distributed (Dedicated))
- Stratosphere (Distributed (Generic))
Setup

- Benchmarking-like experiment
  - 6 algorithms
  - 7 data-sets
  - 7 platforms
- Implement all algorithms on all platforms
- Run and compare ...
  - Performance
- Estimate usability*
Hardware

- DAS4: a multi-cluster Dutch grid/cloud
  - Intel Xeon 2.4 GHz CPU (dual quad-core, 12 MB cache)
  - Memory 24 GB
  - 1 Gbit/s Ethernet network

- Size
  - Most experiments take 20 working machines
  - Up to 50 working machines

- HDFS used as distributed file system
Datasets

<table>
<thead>
<tr>
<th>Graphs</th>
<th>#V</th>
<th>#E</th>
<th>d</th>
<th>(\bar{D})</th>
<th>Directivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1 Amazon</td>
<td>262,111</td>
<td>1,234,877</td>
<td>1.8</td>
<td>4.7</td>
<td>directed</td>
</tr>
<tr>
<td>G2 WikiTalk</td>
<td>2,388,953</td>
<td>5,018,445</td>
<td>0.1</td>
<td>2.1</td>
<td>directed</td>
</tr>
<tr>
<td>G3 KGS</td>
<td>293,290</td>
<td>16,558,839</td>
<td>38.5</td>
<td>112.9</td>
<td>undirected</td>
</tr>
<tr>
<td>G4 Citation</td>
<td>3,764,117</td>
<td>16,511,742</td>
<td>0.1</td>
<td>4.4</td>
<td>directed</td>
</tr>
<tr>
<td>G5 DotaLeague</td>
<td>61,171</td>
<td>50,870,316</td>
<td>2,719.0</td>
<td>1,663.2</td>
<td>undirected</td>
</tr>
<tr>
<td>G6 Synth</td>
<td>2,394,536</td>
<td>64,152,015</td>
<td>2.2</td>
<td>53.6</td>
<td>undirected</td>
</tr>
<tr>
<td>G7 Friendster</td>
<td>65,608,366</td>
<td>1,806,067,135</td>
<td>0.1</td>
<td>55.1</td>
<td>undirected</td>
</tr>
</tbody>
</table>

**Community detection (CD)** is important for social network analysis, as it helps identify communities, that is, groups whose constituent nodes form more than a majority of the vertices. In many graphs, the largest connected component includes a significant portion of the vertices. Community detection algorithms identify cohesive subgraphs, which can be used for applications such as clustering or anomaly detection.

**Connected Component (CONN)** is an algorithm for extracting the largest connected component from a graph. This algorithm is useful for identifying the largest subset of nodes that are mutually reachable, which can be important in applications such as network analysis or social network mining.

**Breadth-first search (BFS)** is a widely used algorithm in graph processing, which is often a building block for more complex algorithms, such as item search, distance calculation, and community detection. BFS is particularly useful for finding the shortest path between two nodes in an unweighted graph.

**STATS** can provide the graph analyst with an overview of the characteristics of a graph, such as the number of vertices and edges, and the average of the local and global degree of vertices. This information is crucial for understanding the structure and properties of a graph, which can inform the choice of algorithms and data models for processing the graph.

**General statistics** (e.g., degree, centrality) are important for understanding the distribution of nodes and edges in a graph. These statistics can be used to identify key nodes or edges in the graph, which can be important for applications such as network analysis or social network mining.

**Vertex neighborhood information** is useful for understanding the connectivity and structure of a graph. This information can be used to identify clusters or communities within a graph, which can be important for applications such as social network analysis or community detection.

**Algorithmic classes** include general statistics, vertex neighborhood information, social network analysis, and community detection, among others. Each class of algorithms is designed to solve specific types of problems, such as finding the largest connected component, identifying communities, or calculating the shortest path between two nodes.
Graph-Processing Algorithms

- Literature survey
  - 10 top research conferences: SIGMOD, VLDB, HPDC ...
  - 2009–2013, 124 articles

<table>
<thead>
<tr>
<th>Class</th>
<th>Examples</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph Statistics</td>
<td>Diameter, PageRank</td>
<td>16.1</td>
</tr>
<tr>
<td>Graph Traversal</td>
<td>BFS, SSSP, DFS</td>
<td>46.3</td>
</tr>
<tr>
<td>Connected Component</td>
<td>Reachability, BiCC</td>
<td>13.4</td>
</tr>
<tr>
<td>Community Detection</td>
<td>Clustering, Nearest Neighbor</td>
<td>5.4</td>
</tr>
<tr>
<td>Graph Evolution</td>
<td>Forest Fire Model, PAM</td>
<td>4.0</td>
</tr>
<tr>
<td>Other</td>
<td>Sampling, Partitioning</td>
<td>14.8</td>
</tr>
</tbody>
</table>
BFS: results for all-2-all

No platform runs fastest for all graphs, but Hadoop is the worst performer. Not all platforms can process all graphs, but Hadoop processes everything.
Giraph: results for all algorithms, all data sets

Storing the whole graph in memory helps Giraph perform well. Giraph may crash when graphs or number of messages large.
Horizontal scalability: BFS on Friendster (31 GB)

Using more computing machines can reduce execution time.

Tuning needed for horizontal scalability, e.g., for GraphLab, split large input files into number of chunks equal to the number of machines.
We need new metrics, to capture meaning of computation time (more later).
In some systems, overhead is by and large wasted time (e.g., in Hadoop).
Additional Overheads: Data ingestion

- Data ingestion
  - Batch system: one ingestion, multiple processing
  - Transactional system: one ingestion, one processing

- Data ingestion matters even for batch systems

<table>
<thead>
<tr>
<th></th>
<th>Amazon</th>
<th>DotaLeague</th>
<th>Friendster</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS</td>
<td>1 second</td>
<td>7 seconds</td>
<td>5 minutes</td>
</tr>
<tr>
<td>Neo4J</td>
<td>4 hours</td>
<td>days</td>
<td>n/a</td>
</tr>
</tbody>
</table>
Productivity

- Low throughput in terms of LOC for all models
- Days to hours development time for the simpler applications

<table>
<thead>
<tr>
<th></th>
<th>Hadoop(Java)</th>
<th>Stratosphere(Java)</th>
<th>Giraph(Java)</th>
<th>GraphLab(C++)</th>
<th>Neo4j(Java)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>1 d, 110 loc</td>
<td>1 d, 150 loc</td>
<td>1 d, 45 loc</td>
<td>1 d, 120 loc</td>
<td>1 h, 38 loc</td>
</tr>
<tr>
<td>CONN</td>
<td>1.5 d, 110 loc</td>
<td>1 d, 160 loc</td>
<td>1 d, 80 loc</td>
<td>0.5 d, 130 loc</td>
<td>1 d, 100 loc</td>
</tr>
</tbody>
</table>

We need better productivity metrics!
Lessons learned*

- Performance is function of (Dataset, Algorithm, Platform, Deployment)
  - Previous performance studies may lead to tunnel vision

- Platforms have their own drawbacks (crashes, long execution time, tuning, etc.)
  - Best-performing is not only low response time
  - Ease-of-use of a platform is very important

- Some platforms can scale up reasonably with cluster size (horizontally) or number of cores (vertically)
  - Strong vs weak scaling still a challenge
    - workload scaling tricky

P-A-D triangle revisited

Algorithm

In progress
Algorithms for different data types and graphs

Overstudied
Performance is enabled
Portability is disabled

In distributed systems, deployment matters, too!

Dataset

Understudied
No systematic findings yet
Intuitive correlations
Must be correlated with the algorithm

Platform
Future directions
Graphalytics*

- Benchmarking graph processing systems
  - Selection of datasets
    - Synthetic, with real profiles
    - Real-life
  - Selection of algorithms
    - Problem-based
    - Code-based
  - Selection of metrics
  - Selection of systems
    - All of them.
  - Reporting the results
    - Goal-oriented

*http://graphalytics.ewi.tudelft.nl
Heterogeneous computing

- Build *the first* multi-node heterogeneous graph processing system
  - Model performance
    - Are GPUs always useful in a distributed setup?
    - Which partition goes where?
  - Implementation
    - Based on existing distributed systems
    - Add graph-specific scheduling and resource allocation
Graph-centric framework

- Understand **graph features**
  - What makes a graph “special”
- Select the best algorithm for a given graph
End goal: Graphitti

1. Understand the hardware
   - HW modeling
     - Generic parameterized hardware model

2. Understand the application
   - Fully Parallel workload model

3. Match hardware to application
   - Hardware configuration
   - Parallel, optimized code
   - Parallelized workload & partitioned dataset
End goal: Graphitti
Take home message

- Graph processing is a hot topic for both software and hardware developers
- Challenges in scale and irregularity
- Existing graph processing systems: 80+
- Choose which one to use
  - Quick-Pick: choose a platform where your graph fits and you can program.
  - Systematic: meta-benchmarking, a.k.a., Graphalytics
Take home message

- Comprehensive and systematic performance study of graph processing systems is difficult.

- Main challenges
  - fairness of comparison
  - development time

- Large-scale systems are promising, but adoption remains low.

- GPU-enabled systems show promising performance, but ... no dedicated distributed GPU-enabled graph processing systems – YET!
Future research directions

- Improved benchmarking
- Heterogeneous computing
- Workload characterization
- Smart resource allocation
Questions?

A.L.Varbanescu@uva.nl
Online: graphalytics.ewi.tudelft.nl