Big Data on Small Machines

Guy Blelloch
with: Aapo Kyrola, Julian Shun

Carnegie Mellon
Question

• There has been a lot of emphasis on large clusters, and tools such as Map Reduce for “big data”

• But, when might it be better to use a smaller machine such as a laptop, desktop, or rack mounted multi-core server?

• Many potential advantages
  – More energy efficient
  – More cost effective
  – Easier to program
  – Easier to administer, reliability
In This Talk

1. Graph analysis using multi-core servers (LIGRA, with Julian Shun, PPOPP ‘13)
2. Graph analysis using a laptops and disks (GraphChi, with Aapo Kyrola, OSDI ’12)
3. In memory compression of graphs, and inverted indices (with Daniel Blandford, SODA ‘04)
Why use Large Clusters

1. Data does not fit in memory of modest machines
2. Modest machines are not fast enough to process the data.
BIG Data

**Sloan Sky Survey:** 7 x $10^{13}$ bytes/year now
7 x $10^{15}$ bytes/year in 2016

**Large Hadron Collider:**
150 million sensors x 40 million samples/sec
= 6 x $10^{16}$ samples/year

**Wallmart Database:** 2.5*10$^{15}$ bytes (predicted)
10 Billion Transactions/year

**YouTube:** 1.2 x $10^8$ videos x 2 x $10^7$ mbytes/video
= 3 x $10^{15}$ bytes

**Genome:** 4 x $10^9$ bp/human x 4 x $10^9$ humans = $10^{19}$ bytes

But most analysis does not have to look at all the data
Large Data (Graphs)

- Web graph (centered around Wikipedia)
## Large Data (Graphs)

<table>
<thead>
<tr>
<th></th>
<th>Edges</th>
<th>Uncompressed</th>
<th>Compressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publicly Available</td>
<td>6 Billion</td>
<td>60 GB</td>
<td>6 GB</td>
</tr>
<tr>
<td>Twitter</td>
<td>25 Billion</td>
<td>250 GB</td>
<td>25 GB</td>
</tr>
<tr>
<td>Facebook</td>
<td>140 Billion</td>
<td>1.4 TB</td>
<td>140 GB</td>
</tr>
<tr>
<td>Web Graph (usefull)</td>
<td>200 Billion?</td>
<td>2 TB</td>
<td>200 Gbytes</td>
</tr>
</tbody>
</table>

6/6/13
Large Text

• Jstor : 2 Million Docs – 100Gbytes
• PubMed 20 Million Docs – 100Gbytes
• Library of congress: $3 \times 10^7$ volumes $\times 10^5$/vol
  3 TB (compressed)

* All numbers estimated
# Machines

<table>
<thead>
<tr>
<th></th>
<th>Cost</th>
<th>Main Memory</th>
<th>Secondary Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laptop</td>
<td>$1K</td>
<td>10 GB</td>
<td>1 TB</td>
</tr>
<tr>
<td>Desktop</td>
<td>$4K</td>
<td>100 GB</td>
<td>10 TB</td>
</tr>
<tr>
<td>Server</td>
<td>$20K</td>
<td>1 TB</td>
<td>100 TB</td>
</tr>
<tr>
<td>100 node Cluster</td>
<td>$200K</td>
<td>10 TB</td>
<td>1000 TB</td>
</tr>
</tbody>
</table>

Twitter Graph: 250 Gbytes
Compressed Twitter Graph: 25 Gbytes
<table>
<thead>
<tr>
<th>Machine</th>
<th>Cost</th>
<th>Main Memory</th>
<th>Secondary Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laptop</td>
<td>$1K</td>
<td>10 GB</td>
<td>1 TB</td>
</tr>
<tr>
<td>Desktop</td>
<td>$4K</td>
<td>100 GB</td>
<td>10 TB</td>
</tr>
<tr>
<td>Server</td>
<td>$20K</td>
<td>1 TB</td>
<td>100 TB</td>
</tr>
<tr>
<td>100 node Cluster</td>
<td>$200K</td>
<td>10 TB</td>
<td>1000 TB</td>
</tr>
</tbody>
</table>

Twitter Graph: 250 Gbytes
Compressed Twitter Graph: 25 Gbytes
# Machines

<table>
<thead>
<tr>
<th>Machine</th>
<th>Cost</th>
<th>Main Memory</th>
<th>Secondary Memory</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laptop</td>
<td>$1K</td>
<td>10 GB</td>
<td>1 TB</td>
<td>4</td>
</tr>
<tr>
<td>Desktop</td>
<td>$4K</td>
<td>100 GB</td>
<td>10 TB</td>
<td>16</td>
</tr>
<tr>
<td>Server</td>
<td>$20K</td>
<td>1 TB</td>
<td>100 TB</td>
<td>64</td>
</tr>
<tr>
<td>100 node Cluster</td>
<td>$200K</td>
<td>10 TB</td>
<td>1000 TB</td>
<td>800</td>
</tr>
</tbody>
</table>

Twitter Graph : 250 Gbytes
Compressed Twitter Graph: 25 Gbytes
In This Talk

1. Graph and Text analysis using multi-core servers (LIGRA, with Julian Shun, PPOPP ‘13)
2. Graph analysis using a laptops and disks (GraphChi, with Aapo Kyrola, OSDI ’12)
3. In memory compression of graphs, and inverted indices (with Daniel Blandford, SODA ‘04)
Ligra

• Lightweight graph processing system for shared memory multicore machines
  – Lightweight: modest amount of code (about 1500 lines including comments)
  – Simple: framework only has 2 routines and one data structure
    • This is enough for a wide class of graph traversal algorithms!
  – Parallel: designed for shared memory systems
  – Efficient: outperform other graph systems by orders of magnitude, up to 39x speedup on 40 cores
  – Designed for synchronous computations

6/6/13

BDA 2013
Ligra Interface

• **Vertex subset**: represents a subset of vertices
  – Important for algorithms that process only a subset of vertices each iteration
  – Can keep around multiple subsets for same graph
  – Can use one subset for multiple graphs

• **Vertex map**: maps user boolean function over vertex subset

• **Edge map**: maps user boolean function over out-edges of vertex subset

• **Note**: system does not store vertex/edge data. User-defined vertex/edge data can be modified by the function passed to map
Parallel Breadth First Search (BFS)
Breadth-first search in Ligra

1: Parents = \{-1, \ldots, -1\} \quad \triangleright \text{initialized to all -1’s}
2:
3: \textbf{procedure} UPDATE(s, d)
4: \quad \textbf{return} \ (\text{CAS}(&\text{Parents}[d], -1, s))
5:
6: \textbf{procedure} COND(i)
7: \quad \textbf{return} \ (\text{Parents}[i] == -1)
8:
9: \textbf{procedure} BFS(G, r) \quad \triangleright \text{r is the root}
10: \quad \text{Parents}[r] = r
11: \quad \text{Frontier} = \{r\} \quad \triangleright \text{vertexSubset initialized to contain only r}
12: \quad \textbf{while} \ (\text{SIZE}(\text{Frontier}) \neq 0) \ \textbf{do}
13: \quad \quad \text{Frontier} = \text{EDGEMAP}(G, \text{Frontier}, \text{UPDATE}, \text{COND})

- compare-and-swap = CAS; takes three arguments \((addr, oldval, newval)\) and atomically updates value at \(addr\) to \(newval\) if its value was \(oldval\). Returns true if updated, false otherwise.
- Line 4 attempts to sets vertex \(s\) to be vertex \(d’\)'s parent if unvisited
- Cond is used to check if unvisited (-1 means unvisited)
- EdgeMap takes frontier, outputs next frontier
Two methods for BFS

Idea due to Beamer, Asanovic and Patterson (2012):

- 1\textsuperscript{st} (Sparse) method better for small frontiers
- 2\textsuperscript{nd} (Dense) method better when frontier is large and many vertices have been visited
- Switch between the two approaches based on frontier size
Experiments

- Used a variety of artificial and real-world graphs
  - Largest is Yahoo web graph with 1.4 billion vertices and 6.6 billion edges
- Implementations in Cilk Plus (extension to C++), 1500 lines of code for the system
- Using 40-core Intel Nehalem based machine
- Good speedup, up to 39x (PageRank)
Comparison to other graph processing systems

<table>
<thead>
<tr>
<th></th>
<th>Ligra: 40 core Performance</th>
<th>vs.</th>
<th>Machines</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth first search</td>
<td>2.5B edges/sec</td>
<td>KDT</td>
<td>64 x 4 core Intel Nehalem</td>
<td>473M edges/sec</td>
</tr>
<tr>
<td>Approximate betweenness centrality</td>
<td>526M edges/sec</td>
<td>KDT</td>
<td>24 x 12 core AMD Opteron processors</td>
<td>125M edges/sec</td>
</tr>
<tr>
<td>Page Rank (15 lines of code)</td>
<td>2.91 sec/iteration for 1.5B edge Twitter graph</td>
<td>Powergraph</td>
<td>8 x 64-core machines</td>
<td>3.6 sec/iteration for 1.5B edge Twitter graph</td>
</tr>
<tr>
<td>Shortest Paths</td>
<td>1.68B edges in under 2 seconds</td>
<td>Pregel</td>
<td>300 multicore commodity PCs</td>
<td>1B edges in 20 seconds</td>
</tr>
</tbody>
</table>

Hadoop: 198sec, Sparc: 97.4sec, Twister 36sec
Ligra - Conclusions

• **Lightweight**: framework is only about 1500 lines of code including comments

• **Simple**: Leads to simple implementations of graph traversal algorithms
  – Abstract main components of graph traversal algorithms into two functions

• **Efficient**: Up to orders of magnitude faster than those of other graph processing systems
  – Not much slower than highly-optimized application-specific code
Ligra - Conclusions

• **Limitations:**
  – Does not work well with dynamic graphs
  – Not for asynchronous computations

• **Future work:**
  – Address limitations
  – Extending to GPUs
In This Talk

1. Graph and Text analysis using multi-core servers (LIGRA, with Julian Shun, PPOPP ‘13)
2. Graph analysis using a laptops and disks (GraphChi, with Aapo Kyrola, OSDI ’12)
3. In memory compression of graphs, and inverted indices (with Daniel Blandford, SODA ‘04)
Vertex-centric Programming

• “Think like a vertex”
• Popularized by the Pregel and GraphLab projects
  – Historically, systolic computation and the Connection Machine

MyFunc(vertex)

{ // modify neighborhood }
Disk-Based Graph Algorithms

**Main Challenge**: Random Access

- 5ms Disk, .05ms SSD, .0001ms memory

**GraphChi Solution**: Parallel Sliding Windows
PSW: Shards and Intervals

- Vertices are numbered from 1 to n
  - P intervals, each associated with a shard on disk.
  - sub-graph = interval of vertices
"I can't see the point of it."
PSW: Layout

Shard: in-edges for interval of vertices; sorted by source-id

Vertices 1..100
Shard 1

Vertices 101..700
Shard 2

Vertices 701..1000
Shard 3

Vertices 1001..10000
Shard 4

Shards small enough to fit in memory; balance size of shards
PSW: Loading Sub-graph

Load subgraph for vertices 1..100

- Vertices 1..100
- Vertices 101..700
- Vertices 701..1000
- Vertices 1001..10000

What about out-edges?
Arranged in sequence in other shards

Load all in-edges in memory
PSW: Loading Sub-graph

Load subgraph for vertices 101..700

Load all in-edges in memory

Vertices 1..100
Shard 1

Vertices 101..700
Shard 2

Vertices 701..1000
Shard 3

Vertices 1001..10000
Shard 4

Out-edge blocks in memory
GraphChi

• C++ implementation: 8,000 lines of code
  – Java-implementation also available (~ 2-3x slower), with a Scala API.
• Several optimizations to PSW (see paper).

Source code and examples:
http://graphchi.org
Evaluation: Is PSW expressive enough?

**Graph Mining**
- Connected components
- Approx. shortest paths
- Triangle counting
- Community Detection

**SpMV**
- PageRank
- Generic

**Recommendations**
- Random walks

**Collaborative Filtering** (by Danny Bickson)
- ALS
- SGD
- Sparse-ALS
- SVD, SVD++
- Item-CF
  + many more

**Probabilistic Graphical Models**
- Belief Propagation

Algorithms implemented for GraphChi (Oct 2012)
Experiment Setting

- Mac Mini (Apple Inc.)
  - 8 GB RAM
  - 256 GB SSD, 1TB hard drive
  - Intel Core i5, 2.5 GHz
- Experiment graphs:

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
<th>P (shards)</th>
<th>Preprocessing</th>
</tr>
</thead>
<tbody>
<tr>
<td>live-journal</td>
<td>4.8M</td>
<td>69M</td>
<td>3</td>
<td>0.5 min</td>
</tr>
<tr>
<td>netflix</td>
<td>0.5M</td>
<td>99M</td>
<td>20</td>
<td>1 min</td>
</tr>
<tr>
<td>twitter-2010</td>
<td>42M</td>
<td>1.5B</td>
<td>20</td>
<td>2 min</td>
</tr>
<tr>
<td>uk-2007-05</td>
<td>106M</td>
<td>3.7B</td>
<td>40</td>
<td>31 min</td>
</tr>
<tr>
<td>uk-union</td>
<td>133M</td>
<td>5.4B</td>
<td>50</td>
<td>33 min</td>
</tr>
<tr>
<td>yahoo-web</td>
<td>1.4B</td>
<td>6.6B</td>
<td>50</td>
<td>37 min</td>
</tr>
</tbody>
</table>
PowerGraph Comparison

• **PowerGraph / GraphLab 2** outperforms previous systems by a wide margin on natural graphs.

• With 64 more machines, 512 more CPUs:
  
  – **Pagerank**: 40x faster than GraphChi  
  – **Triangle counting**: 30x faster than GraphChi.

GraphChi has state-of-the-art performance / CPU.
Hybrid Approach

Possible approach for graphs with metadata:

- Use a server with e.g. 1 TB mem, 100TB Disk
- In memory for main graph
- Disk for meta data
In This Talk

1. Graph and Text analysis using multi-core servers (LIGRA, with Julian Shun, PPOPP ‘13)
2. Graph analysis using a laptops and disks (GraphChi, with Aapo Kyrola, OSDI ’12)
3. In memory compression of graphs, and inverted indices (with Daniel Blandford, SODA ‘04)
Compressing Graphs

• **Goal**: To represent large graphs compactly while supporting queries efficiently
  - e.g., adjacency and neighbor queries
  - want to do significantly better than adjacency lists (e.g. a factor of 10 less space, about the same time)

• **Applications**:
  - Large web graphs
  - Large meshes
  - Phone call graphs
Main Ideas

– Number vertices so adjacent vertices have similar numbers
  • Use separators to do this
– Use difference coding on adjacency lists
– Use efficient data structure for indexing
Renumbering with Edge Separators
Renumbering with Edge Separators
Renumbering with Edge Separators
Renumbering with Edge Separators
Compressed Adjacency Tables

![Graph Diagram]

<table>
<thead>
<tr>
<th>#</th>
<th>D</th>
<th>Neighbors</th>
<th>Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0 2 3</td>
<td>-1 2 1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1 3 4</td>
<td>-1 2 1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1 2 4 5</td>
<td>-1 1 2 1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2 3 5 6</td>
<td>-2 1 2 1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3 4 6 7</td>
<td>-2 1 2 1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>4 5 8</td>
<td>-2 1 3</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>5 8</td>
<td>-2 3</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>6 7</td>
<td>-2 1</td>
</tr>
</tbody>
</table>
Log-sized Codes

• **Log-sized code:** Any prefix code that takes $O(\log(d))$ bits to represent an integer $d$.
• **Gamma code, delta code, skewed Bernoulli code**

**Example: Gamma code**
Prefix: unary code for $\lfloor \log d \rfloor$
Suffix: binary code for $d - 2^\lfloor \log d \rfloor$
(binary code for $d$, except leading 1 is implied)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Gamma</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>00100</td>
</tr>
<tr>
<td>5</td>
<td>00101</td>
</tr>
<tr>
<td>6</td>
<td>00110</td>
</tr>
<tr>
<td>7</td>
<td>00111</td>
</tr>
<tr>
<td>8</td>
<td>0001000</td>
</tr>
</tbody>
</table>
Theorem (edge separators)

• Any class of graphs that allows $O(n^c)$ edge separators can be compressed to $O(n)$ bits with $O(1)$ access time using:
  – Difference coded adjacency lists
  – $O(n)$-bit indexing structure
# Performance: Adjacency Table

<table>
<thead>
<tr>
<th></th>
<th>dfs</th>
<th></th>
<th></th>
<th>bu-bpq</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_d$</td>
<td>Space</td>
<td>$T/T_d$</td>
<td>Space</td>
<td>$T/T_d$</td>
<td>Space</td>
<td>$T/T_d$</td>
<td>Space</td>
<td>$T/T_d$</td>
<td>Space</td>
</tr>
<tr>
<td>auto</td>
<td>0.79</td>
<td>9.88</td>
<td>153.11</td>
<td>5.17</td>
<td>7.54</td>
<td>5.90</td>
<td>14.59</td>
<td>5.52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>feocean</td>
<td>0.06</td>
<td>13.88</td>
<td>388.83</td>
<td>7.66</td>
<td>17.16</td>
<td>8.45</td>
<td>34.83</td>
<td>7.79</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m14b</td>
<td>0.31</td>
<td>10.65</td>
<td>181.41</td>
<td>4.81</td>
<td>8.16</td>
<td>5.45</td>
<td>15.32</td>
<td>5.13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ibm17</td>
<td>0.44</td>
<td>13.01</td>
<td>136.43</td>
<td>6.18</td>
<td>11.0</td>
<td>6.79</td>
<td>20.25</td>
<td>6.64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ibm18</td>
<td>0.48</td>
<td>11.88</td>
<td>129.22</td>
<td>5.72</td>
<td>9.5</td>
<td>6.24</td>
<td>17.29</td>
<td>6.13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td>0.76</td>
<td>8.41</td>
<td>382.67</td>
<td>4.38</td>
<td>14.61</td>
<td>4.90</td>
<td>35.21</td>
<td>4.29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PA</td>
<td>0.43</td>
<td>8.47</td>
<td>364.06</td>
<td>4.45</td>
<td>13.95</td>
<td>4.98</td>
<td>33.02</td>
<td>4.37</td>
<td></td>
<td></td>
</tr>
<tr>
<td>googleI</td>
<td>1.4</td>
<td>7.44</td>
<td>186.91</td>
<td>4.08</td>
<td>12.71</td>
<td>4.18</td>
<td>40.96</td>
<td>4.14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>googleO</td>
<td>1.4</td>
<td>11.03</td>
<td>186.91</td>
<td>6.78</td>
<td>12.71</td>
<td>6.21</td>
<td>40.96</td>
<td>6.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lucent</td>
<td>0.04</td>
<td>7.56</td>
<td>390.75</td>
<td>5.52</td>
<td>19.5</td>
<td>5.54</td>
<td>45.75</td>
<td>5.44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>scan</td>
<td>0.12</td>
<td>8.00</td>
<td>280.25</td>
<td>5.94</td>
<td>23.33</td>
<td>5.76</td>
<td>81.75</td>
<td>5.66</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg</td>
<td></td>
<td>10.02</td>
<td>252.78</td>
<td>5.52</td>
<td>13.65</td>
<td>5.86</td>
<td>34.54</td>
<td>5.56</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time is to create the structure, normalized to time for DFS
Performance: Overall

<table>
<thead>
<tr>
<th>Graph</th>
<th>Array</th>
<th>List</th>
<th>bu-cf/semi</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>space</td>
<td>time</td>
</tr>
<tr>
<td>auto</td>
<td>0.24</td>
<td>34.2</td>
<td>0.61</td>
</tr>
<tr>
<td>feocean</td>
<td>0.04</td>
<td>37.6</td>
<td>0.08</td>
</tr>
<tr>
<td>m14b</td>
<td>0.11</td>
<td>34.1</td>
<td>0.29</td>
</tr>
<tr>
<td>ibm17</td>
<td>0.15</td>
<td>33.3</td>
<td>0.40</td>
</tr>
<tr>
<td>ibm18</td>
<td>0.14</td>
<td>33.5</td>
<td>0.38</td>
</tr>
<tr>
<td>CA</td>
<td>0.34</td>
<td>43.4</td>
<td>0.56</td>
</tr>
<tr>
<td>PA</td>
<td>0.19</td>
<td>43.3</td>
<td>0.31</td>
</tr>
<tr>
<td>googleI</td>
<td>0.24</td>
<td>37.7</td>
<td>0.49</td>
</tr>
<tr>
<td>googleO</td>
<td>0.24</td>
<td>37.7</td>
<td>0.50</td>
</tr>
<tr>
<td>lucent</td>
<td>0.02</td>
<td>42.0</td>
<td>0.04</td>
</tr>
<tr>
<td>scan</td>
<td>0.04</td>
<td>43.4</td>
<td>0.06</td>
</tr>
</tbody>
</table>

*Time is for one DFS*
Conclusions

• For many applications of “large data”, data can fit in the memory of a server or disk of a laptop.
• Speed can be improved on a single multicore server over a distributed system, and significantly more energy efficient.
• Code can be simpler and more general
• Disk algorithms are likely the most energy efficient, so good for high bandwidth embarassigly parallel applications
Thank you!