Algorithms and Tools for Scalable Graph Analytics

Kamesh Madduri
Computer Science and Engineering
The Pennsylvania State University
madduri@cse.psu.edu

MMDS 2012
July 13, 2012
This talk: A methodology for **blazingly fast** graph analytics!
Recent projects involving graph-structured computations

- De novo Genome Assembly [~ 400 GB seq. data]
- Distributed BFS and Graph500 [Synthetic graphs up to a trillion edges]
  - with A. Buluc, SC’11 and DIMACS Challenge 2012
- Indexes for SPARQL queries [~ 100 GB RDF data]
  - with K. Wu, SSDBM ’11
- Parallel centrality computations [graphs with up to 1 billion edges]
  - with M. Frasca and P. Raghavan, SC’12
- Algorithms for $k$ shortest loopless paths [graphs with up to 10 million edges]
0. Having a reference/correct implementation

Possible with well-defined problems

Challenges:
NP-hard problems

Six orders of magnitude performance improvement in the past few years!

And parallel programming challenges ...

And BigData ...
1. Realistic performance estimates

• My new algorithm takes 10 seconds for data set bigdata-x. Is it ‘fast’ and ‘efficient’?

• Given a problem, an algorithm, a data set, and a parallel platform, we need an estimate of execution time.

• Solution: Look beyond asymptotic worst-case analysis
  – Average case analysis
  – Platform-independent algorithm counts
  – Execution time in terms of subroutines/library calls
e.g., A very crude estimate

- ‘Linear work’ algorithm
  - $n = 1$ billion vertices
  - $m = 10$ billion edges
  - Edge represented using 8 bytes

- Lower bound
  - $80 \text{ GB}/(\sim 50 \text{ GB/s} \text{ read BW})$
    - 1.6 seconds
    - 6.25 billion traversed edges per second

AMD ‘Magny-Cours’
e.g., Models designed by computer architects

“Everything should be made as simple as possible, but not simpler.”

Albert Einstein
• Arithmetic Intensity
  – Total floating point operations/total DRAM bytes

• An algorithm/implementation can be
  – Compute-bound, Bandwidth-bound, Latency-bound

• A naïve sparse graph implementation is almost always latency-bound
  – We’d like to make them compute-bound
A simpler estimate of level-synchronous parallel BFS execution time

- The granularity of algorithm analysis I propose:

Intra-node memory cost:

\[ \beta_L \frac{m}{p} + \alpha_{L,n/p} \frac{n + m}{p} \]

- Inverse local RAM bandwidth
- Local latency on working set \(|n/p|\)

Inter-node communication cost:

\[ \beta_{N,a2a}(p) \frac{edgecut}{p} + \alpha_N p \]

- All-to-all remote bandwidth with \(p\) participating processors
2. Data-centric optimizations

We need new algorithms!

- Reducing computation
  - sampling-based methods
  - approximation algorithms
  - Linear-work heuristics

- Improving parallel scaling and efficiency
  - Reducing memory utilization; communication-minimizing graph layout; improving computational load balance

- Orders of magnitude speedup possible!
3. Design memory hierarchy-aware, cache-aware shared memory algorithms

- Move your algorithm from latency-bound to bandwidth-bound regime
- Utilize shared caches more efficiently
- Maximize memory bandwidth
- Strategies
  - Reduce synchronization
  - Reordering
  - Blocked data structures
  - Locality-aware work queues
- Can lead up to an order of magnitude speedup!
Betweenness Centrality

• Shortest paths-based centrality metric
• $O(mn)$ work serial algorithm [Brandes, 2001]
• Our prior work
  – Sampling-based Approximate BC computation
  – Memory-efficient inner-loop parallelization
  – Lock-free strategy
  – Reduction in non-contiguous memory references
• Recent improvements [Frasca et al., SC’12]
  – (Parallel) low-overhead graph reordering
  – NUMA-aware work scheduling
Experimental study: several different graphs

<table>
<thead>
<tr>
<th>Name</th>
<th>DIMACS</th>
<th>LAW</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1: germany_osm</td>
<td>11.5</td>
<td>18.5</td>
</tr>
<tr>
<td>D2: asia_osm</td>
<td>12.0</td>
<td>22.7</td>
</tr>
<tr>
<td>D3: hugetrace-00010</td>
<td>12.1</td>
<td>39.5</td>
</tr>
<tr>
<td>D4: road_central</td>
<td>14.1</td>
<td>41.3</td>
</tr>
<tr>
<td>D5: hugetrace-00020</td>
<td>16.0</td>
<td>50.6</td>
</tr>
<tr>
<td>D6: nlpkkt200</td>
<td>16.2</td>
<td>28.0</td>
</tr>
<tr>
<td>D7: rgg_n_2_24_s0</td>
<td>16.8</td>
<td>23.9</td>
</tr>
<tr>
<td>D8: delaunay_n24</td>
<td>16.8</td>
<td>28.9</td>
</tr>
<tr>
<td>D9: hugebubbles-00000</td>
<td>18.3</td>
<td>28.0</td>
</tr>
<tr>
<td>D10: hugebubbles-00010</td>
<td>19.5</td>
<td>28.0</td>
</tr>
<tr>
<td>D11: hugebubbles-00020</td>
<td>21.2</td>
<td>28.0</td>
</tr>
<tr>
<td>D12: road_usa</td>
<td>23.9</td>
<td>28.9</td>
</tr>
<tr>
<td>D13: nlpkkt240</td>
<td>28.0</td>
<td>23.9</td>
</tr>
<tr>
<td>D14: europe_osm</td>
<td>50.9</td>
<td>50.9</td>
</tr>
</tbody>
</table>

Performance results on a quad-socket Intel Westmere-EX server
- Xeon E7-8837 processors
- 2.6 GHz processor
- 24 MB L3 cache per processor
- 256 GB memory

(vertex and edge counts in millions)
Enhanced scaling on NUMA platforms
Optimizations enable reduced per-thread working sets ...
and improve cache hit rates
4. Space/time tradeoffs

• HPC systems typically provide greater amounts of fast memory
  – Use it (judiciously)

• Some good examples
  – Preprocessing to store auxiliary information
  – replicate shared data structures
  – data layout to minimize inter-node communication., i.e., replication

• Bad example
  – Beefy in-memory representation
5. Not all big data problems are big graph problems: turn yours into a ‘small graph’ problem

- May want to operate on local structure, an induced subgraph with vertices of interest
- Exploit hierarchical structure in networks
- Sparsify networks
- Divide and conquer

- Genome assembly: sequence data can be reduced into a graph problem that is two orders of magnitude smaller
De novo Genome Assembly

ACACGTGTGCACTACTGCACTCTACTCCACTGACTA

Short reads
~ 100’s GB/run

“Scaffold” the contigs

Genome assembler

Align the reads

Genome

nucleotide

~ billions of nucleotides

Sample

Sequencer

~ PB/yr

Contigs

Genome assembler
De novo Metagenome Assembly

Short reads
~ 100’s GB/run

"Scaffold" the contigs

Parallel metagenome assembly

De Bruijn graph

100’s – 1000’s of organisms
~ millions of bases each

Sample

Sequencer
~ PB/yr
6. Partition, if you must

• High-dimensional data, lack of balanced separators

• Implications for memory-intensive graph computations
  – \( O(m) \) inter-node communication, \( O(m) \) local memory references
  – network bandwidths/latencies will be the primary performance limiters

• Load balancing is non-trivial
  – the ‘trivial’ solution is randomly shuffling vertex identifiers, but that destroys locality
Parallel BFS (from a single vertex) on a static, undirected synthetic network (R-MAT generator) with average vertex degree 16.

- Evaluation criteria:
  - minimum execution time
  - largest problem size

- Reference distributed and shared memory implementations provided.

- Computers/supercomputers ranked every 6 months
Graph500 Performance: top parallel system

- IBM BlueGene/P: 8192 nodes
- ANL Mira: IBM BlueGene/Q, 32768 nodes
- Best single-node performance: Convey HC-2EX, 7.85 GTEPS
Graph500 Normalized Performance (per node)
Graph500 Normalized Performance (per node)

Top entries

500 nodes of Cray XT4
MPI-only
All-to-all communication limited performance
Competition ranking criterion: largest problem size
Graph500 Normalized Performance (per node)

- Millions of traversed edges/second
- Top entries
- LBNL/NERSC
- Submissions
- 1800 nodes of Cray XE6
  MPI + OpenMP
  All-to-all communication limited performance
  Competition ranking criterion:
  largest problem size
Graph500 Normalized Performance (per node)

- Millions of traversed edges/second
- Top entries
- LBNL/NERSC submissions

4000 nodes of Cray XE6
MPI + OpenMP

Competition ranking criterion:
peak performance => Smaller problem size
2D partitioning
Led to a 2X performance improvement
Graph500 Normalized Performance (per node)

- **Millions of traversed edges/second**
- **Top entries**
- **Ninja programming!**
- **LBNL/NERSC Submissions**
- **#2**
- **#2**
- **#2**
- **#8**

4817 nodes of Cray XE6
Heuristic to reduce memory references in power-law graphs
[Beamer et al., 2011]
Again, led to a **2X performance improvement**
Balanced partitions, reduced edge cut does not necessarily mean faster graph algorithm execution

- Execution timeline for parallel BFS on a web crawl (eu-2005)
- 16 nodes of Cray XE6 (times with 4 nodes shown below)
7. Adapt existing scalable frameworks/tools for your problem

• Problems amenable to a MapReduce-style of execution
• Borrow ideas from scientific computing, particularly parallel sparse linear algebra

• Our recent work: Adapting FastBit, a compressed bitmap index, to speed up SPARQL queries
Semantic data analysis and RDF

• The RDF (Resource Description Framework) data model is a popular abstraction for linked data repositories
  – Records in *triple* form [<subject> <predicate> <object>]
  – Data sets with a few billion triples quite common

• **Triple-stores**: custom databases for storage and retrieval of RDF data
  – Jena, Virtuoso, Sesame
SPARQL

- Query language expressing **conjunctions** and disjunctions of triple patterns
- Each conjunction corresponds to a database join
- SPARQL queries can be viewed as **graph pattern-matching** problems

- Example query from the Lehigh University Benchmark Suite (LUBM):
  ```sparql
  select ?x ?y ?z where {
    ?x rdf:type ub:GraduateStudent .
    ?y rdf:type ub:University .
    ?z rdf:type ub:Department .
    ?x ub:memberOf ?z .
    ?x ub:undergraduateDegreeFrom ?y .
  }
  ```
FastBit+RDF: Our Contributions

• We use the compressed bitmap indexing software **FastBit** to index RDF data
  – Several different types of bitmap indexes
  – Scalable parallel index construction

• We present a new SPARQL query evaluation approach
  – Pattern-matching queries on RDF data are modified to use bitmap indexes

• Our approach is **1.4-13X** faster than the **RDF-3X** SPARQL query software
  – Speedup insight: The nested joins in SPARQL queries can be expressed as fast bitvector operations.
8. Always question ‘conventional wisdom’

- With appropriate sanity checks
  - i.e., $O(n^2)$ algorithms aren’t a good idea for massive data, even on massively parallel systems

- Several innovative ideas from this workshop
Summary: Methodology for High-performance large graph analytics

1. Performance Models
2. Data-centric alg.
3. Memory opt.
4. Space/time tradeoffs
5. Reduce problem size
6. Scale out
7. Adapt current state-of-the-art tools

My recent research contributions

• Parallel Centrality
• Genome assembly
• Parallel BFS
• SPARQL queries
Acknowledgments

• M. Frasca, P. Raghavan
• M. Poss, M. Roossinck
• A. Buluc
• K. Wu, S. Williams, L. Oliker
• V. Markowitz, K. Yelick, R. Egan
Thank you!

• Questions?
  – madduri@cse.psu.edu
  – madduri.org