The BFS Kernel: Applications and Implementations

Peter M. Kogge
McCourtney Prof. of CSE
Univ. of Notre Dame
IBM Fellow (retired)

Some Interesting Applications

- Six Degrees of Kevin Bacon
- From: https://www.geeksforgeeks.org/applications-of-breadth-first-traversal/
  - Search for neighbors in peer-peer networks
  - Search engine web crawlers
  - Social networks – distance k friends
  - GPS navigation to find “neighboring” locations
  - Patterns for “broadcasting” in networks
  - Community Detection
  - Maze running
  - Routing of wires in circuits
  - Finding Connected components
  - Copying garbage collection, Cheney’s algorithm
  - Shortest path between two nodes u and v
  - Cuthill–McKee mesh numbering
  - Maximum flow in a flow network
  - Serialization/Deserialization of a binary tree
  - Construction of the failure function of the Aho-Corasick pattern matcher.
  - Testing bipartiteness of a graph
Key Kernel: BFS - Breadth First Search

- Given a huge graph
- Start with a root, find all reachable vertices
- Performance metric: TEPS: Traversed Edges/sec

```
Starting at 1: 1, 0, 3, 2, 9, 5
```

No Flops – just Memory & Networking

Definitions

- **Graph** $G = (V, E)$
  - $V = \{v_1, \ldots, v_M\}$, $|V| = N$
  - $E = \{(u, v)\}$, $u$ and $v$ are vertices, $|E| = M$
- **Scale:** $\log_2(N)$
- **Out-degree:** # of edges leaving a vertex
- "Heavy" vertex: has very large out-degree
  - $H = $ subset of heavy vertices from $V$
- **Node:** standalone processing unit
- **System:** interconnected set of P nodes
- **TEPS:** Traversed Edges per Second
**Notional Sequential Algorithm**

- **Forward search:** Keep a “frontier” of new vertices that have been “touched” but not “explored”
  - Explore them and repeat
- **Backward search:** look at all “untouched vertices” and see if any of their edges lead to a touched vertex
  - If so, mark as touched, and repeat
- **Special considerations**
  - Vertices that have huge degrees

**Notional Data Structures**

- **Vis** = set of vertices already “visited”
  - Initially just root $v_s$
- **In** = “Frontier”
  - subset of Vis reached for 1st time on last iteration
- **Out** = set of previously untouched vertices that have 1 edge from frontier
- **P[v]** = “predecessor” or “parent” of $v$
Sequential “Forward” BFS:
Explore forward from Frontier

\[
\text{while } |\text{In}| \neq 0 \\
\text{Out} = \{\}; \\
\text{for } u \text{ in In do} \\
\text{for } v \text{ in some edge } (u,v) \\
\quad \text{if } v \text{ not in Vis} \\
\quad \quad \text{Out} = \text{Out} \cup \{v\}; \\
\quad \quad \text{Vis} = \text{Vis} \cup \{v\}; \\
\quad \quad \text{P}[v] = u; \\
\text{In} = \text{Out};
\]

From each vertex in frontier
follow each edge
and if untouched, add to new frontier

Block executed for each edge traversed

TEPS = # of times/sec this block is executed

Sequential “Backward” BFS
Explore backwards from Untouched

\[
\text{while vertices were added in prior step} \\
\text{Out} = \{\}; \\
\text{for } v \text{ not in Vis do} \\
\text{for } u \text{ in some edge } (u,v) \\
\quad \text{if } u \text{ in Vis} \\
\quad \quad \text{Out} = \text{Out} \cup \{v\}; \\
\quad \quad \text{Vis} = \text{Vis} \cup \{v\}; \\
\quad \quad \text{P}[v] = u;
\]
Key Observation

- Forward direction requires investigation of every edge leaving a frontier vertex
  - Each edge can be done in parallel
- Backwards direction can stop investigating edges as soon as 1 vertex in current frontier is found
  - If search edges sequentially, potentially significant work avoidance
- In any case, can still parallelize over vertices in frontier

Beamer’s Hybrid Algorithm

- Switch between forward & backward steps
  - Use forward iteration as long as In is small
  - Use backward iteration when Vis is large
- Advantage: when
  - # edges from vertices in !Vis
  - are less than # edges from vertices in In
  - then we follow fewer edges overall
- Estimated savings if done optimally: up to 10X reduction in edges
Edges Explored per Level

By this level, most vertices now touched, so edges explored mostly point backward

Few nodes in early levels mean few edges

Going backwards from untouched vertices, and stopping on first touch, reduces # edges covered to near-optimal (optimal is 1 edge per vertex)

Few nodes in early levels mean few edges

Notes

• TEPS is computed as # edges in connected component / execution time
  – Property of graph, not algorithm
  – Thus traversing same edge >1 only counts as 1 time
  – And not traversing an edge still counts as 1

Fig. 5: Graph properties at each exploration level.
Checconi and Petrini, “Traversing Trillions …”
Graph500

Graph500: www.graph500.org

- Several years of reports on performance of BFS implementations on
  - Different size graphs
  - Different hardware configurations
- Standardized graphs for testing
- Standard approach for measuring
  - Generate a graph of certain size
  - Repeat 64 times
    - Select a root
    - Find “level” of each reachable vertex
    - Record execution time
    - TEPS = graph edges / execution time
Graph500 Graphs

- Kronecker graph generator algorithm
  - D. Chakrabarti, Y. Zhan, and C. Faloutsos, R-MAT: A recursive model for graph mining, SIAM Data Mining 2004

- Recursively sub-divides adjacency matrix into 4 partitions A, B, C, D

- Add edges one at a time, choosing partitions probabilistically
  - A = 57%, B = 19%, C = 19%, D = 5%

- # of generated edges = 16*# vertices
  - Average Vertex Degree is 2X this

Graph Sizes

<table>
<thead>
<tr>
<th>Level</th>
<th>Scale</th>
<th>Size</th>
<th>Vertices (Billion)</th>
<th>TB</th>
<th>Bytes /Vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>26</td>
<td>Toy</td>
<td>0.1</td>
<td>0.02</td>
<td>281.8048</td>
</tr>
<tr>
<td>11</td>
<td>29</td>
<td>Mini</td>
<td>0.5</td>
<td>0.14</td>
<td>281.3952</td>
</tr>
<tr>
<td>12</td>
<td>32</td>
<td>Small</td>
<td>4.3</td>
<td>1.1</td>
<td>281.472</td>
</tr>
<tr>
<td>13</td>
<td>36</td>
<td>Medium</td>
<td>68.7</td>
<td>17.6</td>
<td>281.4752</td>
</tr>
<tr>
<td>14</td>
<td>39</td>
<td>Large</td>
<td>549.8</td>
<td>141</td>
<td>281.475</td>
</tr>
<tr>
<td>15</td>
<td>42</td>
<td>Huge</td>
<td>4398.0</td>
<td>1,126</td>
<td>281.475</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Average 281.5162</td>
</tr>
</tbody>
</table>

Scale = log₂(# vertices)
Available Reference Implementations

- Sequential
- Multi-threaded: OPENMP, XMP
- Distributed using MPI
  - Distribute vertices among nodes, including edge lists
  - Each node keeps bit vectors of its vertices
    - One vector of "touched"
    - Two vectors of "frontier" – current and next
  - For each level, all nodes search their current frontiers
    - For each vertex, send message along each edge
      - If destination vertex is "untouched", mark as touched and mark next frontier
    - At end of levels make next frontier the current frontier

Graph500 Report Analysis
Goal

- Match Graph 500 reports with actual hardware
- Correlate performance with hardware & system parameters
  - Hardware: Core type, Peak flops, bandwidth, ...
  - System: System architecture, ...
- Look at results thru lens of architectural parameters
- Do so in way that allows apples-apples across benchmarks
- Note: not all current reports fully correlated

Units of Parallelism

- **Cores**: can execute independent threads
- **Sockets**: contain multiple cores
- **Node**: minimal unit of sockets & memory
- **Endpoint**: set of nodes visible to network as single unit
- **Blade**: physical block of $\geq 1$ endpoints
- **Rack**: Collection of blades
- **Domain**: set of cores that share same address space, all accessible via load/stores
## 2D Architectural Classification

<table>
<thead>
<tr>
<th>System Interconnect</th>
<th>Core Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L</strong>: Loosely coupled distributed memory</td>
<td><strong>H</strong>: Heavyweight</td>
</tr>
<tr>
<td>- Commodity networking with software I/F</td>
<td><strong>L</strong>: Lightweight</td>
</tr>
<tr>
<td><strong>T</strong>: Tightly coupled distributed memory</td>
<td><strong>B</strong>: BlueGene</td>
</tr>
<tr>
<td>- Specialized NICs &amp; some H/W RDMA ops</td>
<td><strong>X</strong>: Multi-threaded</td>
</tr>
<tr>
<td><strong>S</strong>: Shared Memory</td>
<td><strong>V</strong>: Vector</td>
</tr>
<tr>
<td>- Single domain in H/W</td>
<td><strong>O</strong>: Other</td>
</tr>
<tr>
<td><strong>D</strong>: Distributed Shared Memory</td>
<td><strong>G</strong>: GPU-like</td>
</tr>
<tr>
<td>- Single domain but S/W assist for remote references (typically via traps)</td>
<td><strong>M</strong>: a mix</td>
</tr>
</tbody>
</table>

### Examples

<table>
<thead>
<tr>
<th>System Interconnect</th>
<th>Core Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T</strong>: Tightly coupled</td>
<td><strong>H</strong>: Heavyweight: Xeon</td>
</tr>
<tr>
<td>- Cray systems with Aries NICs</td>
<td><strong>L</strong>: Lightweight: ARM</td>
</tr>
<tr>
<td><strong>L</strong>: Loosely coupled</td>
<td><strong>B</strong>: BlueGene</td>
</tr>
<tr>
<td>- Infiniband Networking</td>
<td><strong>X</strong>: Multi-threaded: XMT</td>
</tr>
<tr>
<td><strong>S</strong>: Shared Memory</td>
<td><strong>V</strong>: Vector: NEC SX</td>
</tr>
<tr>
<td>- SGI UV systems, XMT</td>
<td><strong>O</strong>: Other: Convey</td>
</tr>
<tr>
<td><strong>D</strong>: Dist. Shared Memory</td>
<td><strong>G</strong>: GPU-like: Nvidia</td>
</tr>
<tr>
<td>- Numascale</td>
<td><strong>M</strong>: a mix</td>
</tr>
</tbody>
</table>
A Modern “Multi-Node” Endpoint

Each Node a Separate Domain

Conventional Processing Node

Domains can span Endpoints if NIC in Memory Path

HPL Architectural Change

Significant changes in Architecture Over Time
Key Architectural Parameters

- $R_{\text{peak}}$: peak flop rate
- Memory bandwidth: peak data exchange rate between memory chips & socket(s)
- Memory Access Rate: peak # of random, independent memory accesses per second
- Peak Network Injection Bandwidth (for tight or loosely coupled)
- # Cores, Sockets, Nodes, Endpoints, Domains, Blades, Racks
- Total Memory Capacity
- Total Power

BFS Over Time

Not A Lot of Growth at Top
Massive dropoff with multiple domains
Sockets in Shared Memory Systems seem to give best Less Efficient
**Performance vs # Local Vertices**

More Vertices storable on each node increases performance

1. **E** - 05
2. **E** - 04
3. **E** - 03
4. **E** - 02
5. **E** - 01
6. **E** + 00
7. **E** + 01
8. **E** + 02
9. **E** + 03
10. **E** + 04
11. **E** + 05
12. **E** + 06
13. **E** + 07
14. **E** + 08
15. **E** + 09
16. **E** + 10
17. **E** + 11
18. **E** + 12

**Sparsity & Parallelism**

Observation: Extreme Sensitivity to
- Level of Sparsity
- # of physically separate memory domains

Across all kernels, it takes 10-1000 nodes of distributed memory systems to equal best of single domain systems for the sparsest problems
Conclusions

- 3 Performance regions
  - Single Domain: highest performance per core, ... – by far
  - < 1 Rack
    - Significant drop-off from single domain
    - But excellent weak scaling
    - Especially shared memory vector machines
  - > 1 Rack
    - Another drop-off from single rack
    - But again good scaling up to about 1 million cores

- Strong correlation with memory bandwidth
  - But Shared Memory more effective using bandwidth

- Strongly invite more “low parallelism” reports
Blue Gene Q Implementations

GTEPS vs Node Count: All Systems

Performance almost linear in # nodes over 1000 nodes

Except at single node
The best nodes are improving

GTEPS/Node: BlueGene Only

BB/P  BG/Q
Recent BG/Q Measurements

<table>
<thead>
<tr>
<th>Date</th>
<th>Scale</th>
<th>GTEPS</th>
<th>Number of Nodes</th>
<th>Memory (GB)</th>
<th>GTEPS/Node</th>
<th>Vertices</th>
<th>GTEPS/Vertex</th>
<th>Cache Bit/Vertex</th>
<th>Mem. Bytes/Vertex</th>
<th>Memory BW (GB/s)</th>
<th>TEP</th>
<th>Accesses per TEP</th>
</tr>
</thead>
<tbody>
<tr>
<td>11/1/2014</td>
<td>33</td>
<td>172</td>
<td>512</td>
<td>8,192</td>
<td>3.36E+01</td>
<td>8.6E+09</td>
<td>1.6E+07</td>
<td>16.00</td>
<td>1024</td>
<td>127</td>
<td>0.99</td>
<td></td>
</tr>
<tr>
<td>11/1/2014</td>
<td>34</td>
<td>294</td>
<td>1024</td>
<td>16,384</td>
<td>2.87E+01</td>
<td>1.7E+10</td>
<td>1.6E+07</td>
<td>16.00</td>
<td>1024</td>
<td>148</td>
<td>1.16</td>
<td></td>
</tr>
<tr>
<td>11/1/2014</td>
<td>34</td>
<td>382</td>
<td>1024</td>
<td>16,384</td>
<td>3.73E+01</td>
<td>1.7E+10</td>
<td>1.6E+07</td>
<td>16.00</td>
<td>1024</td>
<td>114</td>
<td>0.89</td>
<td></td>
</tr>
<tr>
<td>11/1/2014</td>
<td>35</td>
<td>769</td>
<td>2048</td>
<td>32,768</td>
<td>3.75E+01</td>
<td>3.4E+10</td>
<td>1.6E+07</td>
<td>16.00</td>
<td>1024</td>
<td>114</td>
<td>0.88</td>
<td></td>
</tr>
<tr>
<td>7/8/2015</td>
<td>36</td>
<td>0.601</td>
<td>64</td>
<td>1024</td>
<td>9.40E+00</td>
<td>6.9E+09</td>
<td>1.07E+09</td>
<td>25.00</td>
<td>16</td>
<td>4541</td>
<td>35.34</td>
<td></td>
</tr>
<tr>
<td>11/1/2014</td>
<td>36</td>
<td>1427</td>
<td>4096</td>
<td>65,536</td>
<td>3.48E+01</td>
<td>6.9E+10</td>
<td>1.6E+07</td>
<td>16.00</td>
<td>1024</td>
<td>122</td>
<td>0.95</td>
<td></td>
</tr>
<tr>
<td>11/1/2014</td>
<td>37</td>
<td>2567</td>
<td>8192</td>
<td>131,072</td>
<td>3.13E+01</td>
<td>1.4E+11</td>
<td>1.6E+07</td>
<td>16.00</td>
<td>1024</td>
<td>136</td>
<td>1.06</td>
<td></td>
</tr>
<tr>
<td>11/1/2014</td>
<td>38</td>
<td>5848</td>
<td>16384</td>
<td>262,144</td>
<td>3.57E+01</td>
<td>2.7E+11</td>
<td>1.6E+07</td>
<td>16.00</td>
<td>1024</td>
<td>120</td>
<td>0.93</td>
<td></td>
</tr>
<tr>
<td>11/1/2014</td>
<td>39</td>
<td>14892</td>
<td>49152</td>
<td>786,432</td>
<td>3.05E+01</td>
<td>1.1E+12</td>
<td>2.24E+07</td>
<td>12.00</td>
<td>768</td>
<td>140</td>
<td>1.09</td>
<td></td>
</tr>
<tr>
<td>11/1/2014</td>
<td>40</td>
<td>23751</td>
<td>98304</td>
<td>1,572,860</td>
<td>2.42E+01</td>
<td>2.2E+12</td>
<td>2.24E+07</td>
<td>12.00</td>
<td>768</td>
<td>177</td>
<td>1.37</td>
<td></td>
</tr>
</tbody>
</table>

**Observations**

- **Blue**: Highest GTEPS per node
  - 0.375 GTEPS, 16M vertices /node
- **Orange**: Highest vertices per node
  - 1.07B vertices but only 0.0094 GTEPS /node
- **Red**: Highest overall GTEPS & biggest scale
  - But only 0.24 GTEPS, 22.4M vertices /node

TEPS vs # Racks of Q (Blue #s)

Measurements made over 64 runs
- Run with max performance
- 1st Quartile
- Median
- 3rd Quartile
- Run with min performance

After 16 racks edge distribution imbalance increases, causing reduced scaling.
Message Passing

- **Forward direction:**
  - Node $n_i$ sends a message to each node $n_j$ where
    - Some vertex $u$ is owned by $n_i$, and $u$ is currently in $In$
    - And there is some edge $(u,v)$ and $v$ is owned by $n_j$

- **Backward direction:**
  - Node $n_i$ sends a message to each node $n_j$ where
    - Some vertex $v$ is owned by $n_i$, and $v$ is currently not in $In$
    - And there is some edge $(u,v)$ and $u$ is owned by $n_j$
  - If that message finds a $u$ that is in $In$
    - Then reply message sent back to node $n_i$ to update $v$

Distributed Data Decomposition

- How are vertices and edges distributed in parallel system
  - **1D**: Each node owns subset of vertices
    - If $u$ is on $n_j$, so are all edges $(u,v)$
    - Problem: when $u$ has very high out-degree
  - **2D**: Each node owns subset of edges
    - Equivalent to owning all edges between subsets $V_i$ and $V_j$ of vertices
    - Better distribution of edges for heavy vertices
BlueGene Q 1D Algorithm:
Most TEPS/Node for BG/Q

<table>
<thead>
<tr>
<th>Date</th>
<th>MTEPS</th>
<th>GTEPS</th>
<th>Number of Nodes</th>
<th>Memory (GB)</th>
<th>Vertices</th>
<th>Vertices/Node</th>
<th>Memory BB/Vert</th>
<th>Mem BB/Vert (bytes)</th>
<th>MM Accesses/TEP</th>
</tr>
</thead>
<tbody>
<tr>
<td>11/1/2014</td>
<td>35</td>
<td>172</td>
<td>512</td>
<td>8,192</td>
<td>3.36E+07</td>
<td>8.6E+09</td>
<td>16.00</td>
<td>120</td>
<td>0.99</td>
</tr>
<tr>
<td>11/1/2014</td>
<td>36</td>
<td>304</td>
<td>1024</td>
<td>16,384</td>
<td>1.76E+09</td>
<td>1.68E+07</td>
<td>16.00</td>
<td>120</td>
<td>1.36</td>
</tr>
<tr>
<td>11/1/2014</td>
<td>38</td>
<td>382</td>
<td>1024</td>
<td>16,384</td>
<td>1.76E+09</td>
<td>1.68E+07</td>
<td>16.00</td>
<td>118</td>
<td>0.89</td>
</tr>
<tr>
<td>11/1/2014</td>
<td>39</td>
<td>769</td>
<td>2048</td>
<td>32,768</td>
<td>2.87E+09</td>
<td>1.68E+07</td>
<td>16.00</td>
<td>118</td>
<td>0.89</td>
</tr>
<tr>
<td>7/8/2015</td>
<td>20</td>
<td>601</td>
<td>64</td>
<td>1.10E+09</td>
<td>1.68E+07</td>
<td>16.00</td>
<td>120</td>
<td>0.93</td>
<td></td>
</tr>
<tr>
<td>11/1/2014</td>
<td>36</td>
<td>1427</td>
<td>4096</td>
<td>65,536</td>
<td>7.65E+07</td>
<td>1.07E+09</td>
<td>16.00</td>
<td>120</td>
<td>0.95</td>
</tr>
<tr>
<td>11/1/2014</td>
<td>37</td>
<td>1551</td>
<td>8192</td>
<td>131,072</td>
<td>3.73E+07</td>
<td>1.68E+07</td>
<td>16.00</td>
<td>120</td>
<td>1.06</td>
</tr>
<tr>
<td>11/1/2014</td>
<td>38</td>
<td>3848</td>
<td>16384</td>
<td>262,144</td>
<td>1.35E+07</td>
<td>1.68E+07</td>
<td>16.00</td>
<td>120</td>
<td>0.93</td>
</tr>
<tr>
<td>11/1/2014</td>
<td>40</td>
<td>14982</td>
<td>99532</td>
<td>786,432</td>
<td>1.35E+07</td>
<td>1.68E+07</td>
<td>16.00</td>
<td>120</td>
<td>1.09</td>
</tr>
<tr>
<td>11/1/2014</td>
<td>41</td>
<td>23751</td>
<td>98004</td>
<td>1,572,860</td>
<td>2.42E+07</td>
<td>2.24E+07</td>
<td>12.00</td>
<td>768</td>
<td>1.37</td>
</tr>
</tbody>
</table>

BlueGene/Q Data Distribution

- Each node owns subset of vertices
- Non-heavy vertices \( \{u\} \): 1D distribution of edges
  - All edges \((u, v)\) from \(u\) stored on owner\((u)\)
- Heavy vertices \( \{h\} \):
  - Edges distributed throughout system
  - With \( \{h, v\} \) stored on owner\((v)\)
Data Structures

- **In, Out, Vis**: all bit vectors
  - 1 bit per “non-heavy” vertex
  - With node $n_i$ holding bits for all/only vertices it owns
- **In$_i$, Out$_i$, Vis$_i$** refer to part held by node $i$
- **P**: array with one # per vertex
  - $P[v] =$ vertex number of predecessor of $v$
  - Partitioned so $P[v]$ on node that owns $v$
- **In$^H$, Out$^H$, Vis$^H$** all bit vectors for heavies
  - Complete copies In$^H_n$, Out$^H_n$, Vis$^H_n$ on each node $n$
  - Likewise $P^H_n$ is separate copy on node $n$

Non-Heavy Edges

- Each node holds combined edge list for its owned vertices in single array in CSR format
- **Edge sub-list for one non-heavy vertex**
  - Source vertex number stored in 64bit word
    - actually offset within local’s range ($<<40$ bits)
    - With remaining bits an offset to start of edge list for next local vertex
  - List of destination vertex numbers
    - 40 bits each in a 64 bit word
    - If vertex is heavy, upper 24 bits are index into $H$
- **Coarse Index Array**
  - One entry for every 64 local vertices points to start in CSR array
  - To find vertex $64k+j$, start at kth index & search
  - 64 chosen to match 64 bits of bit vectors
while In != {} do
    dir = CalculateDirection;
    if dir = FORWARD
        for u in In do
            for v such that (u,v) in E do
                send(u, v, FORWARD) to owner(v);
    else
        for v not in In do
            for u such that (u,v) in E do
                send(u, v, BACKWARD) to owner(u);
    In = Out;

Function Receive(u, v, dir)
    if dir = FORWARD
        if v not in Vis
            Vis = Vis U {v};
            Out = Out U {v};
            P[v] = u;
        else if u in In
            send(u, v, FORWARD) to owner(v);

Forward Step for Heavies
Out = {}
for u in In do
    for each v from (u, v) in E do
        if v in H then
            Vis = Vis U {v};
            Out = Out U {v};
            P[v] = u;
        else
            Vis = Vis U {v};
            Out = Out U {v};
            P[v] = u;
        allreduce (Vis, OR);
        allreduce (Out, OR);
        In = Out;
Backward Step for Heavies

Out^H_n = {}; All nodes look at all heavies
for v in ~Vis^H_n do But only process untouched heavies
   for each (u, v) in E_n do
      if u in H then
         if u in In^H do
            Vis^H_n = Vis^H_n U {v}; If source of edge is heavy
            Out^H_n = Out^H_n U {v}; then update local copy
            P^H_n[v] = u; of heavy data structures
         else if u in In_n do
            Vis_n = Vis_n U {v}; If source is not heavy but
            Out_n, = Out_n U {v}; local, then update local copy
            P_n[v] = u; of non-heavy data structures.
            allreduce (Vis^H_n, OR); non-local non-heavy source
            allreduce (Out^H_n, OR); handled by other loop
            In^H = Out^H_n; Note! no messages needed in loop!!!

Message Packing

- Each send uses target node to identify a local buffer (need 1 buffer per node)
- Message is placed in that buffer until it is full
- When full, buffer is sent as single packet to target
- Target unpacks the packet and performs series of receives
- Packet format
  - Header ~8B identifying source id and size of rest
  - At most 6 bytes for each (u, v) pair
    - 24 bits for source local index (with rest of 40 bit index from source node id)
    - 24 bits for target local index (we know upper 16 bits are that associated with this node)
  - When possible use only 4 bytes per pair
    - 24 bits for source vertex
    - 7 bits as a difference from last target vertex # in this packet
BlueGene/Q Analysis: “Blue” Algorithm

BlueGene/Q Node

- 16-core logic chip, each core:
  - 1.6GHz, 4-way multi-threaded
  - 16KB L1 data cache with 64B lines, 16KB L1 instruction
  - 8 DP flops per cycle = 12.8 Gflops/sec per core
- 32MB Shared L2
  - 16 2MB sections
  - Rich set of atomic ops at L2 interface
    - Up to 1 every 4 core cycles per section
    - Load, Load&Clear, Load&Increment, Load&Decrement
    - LoadIncrementBounded & LoadDecrementBounded
      - Assumes 8B counter at target and 8B bound in next location
    - StoreAdd, StoreOR, StoreXor combines 8B data into memory
    - StoreMaxUnsigned, StoreMaxSigned
    - StoreAddCoherenceOnZero
    - StoreTwin stores value to address and next, if they were equal
BlueGene/Q Node (Continued)

- 2 DDR3 memory channels, each
  - 16B+ECC transaction width, 1.333GT/s
  - 21.33 GB/s, 0.166B accesses per second, each returning 128B

- 10+1 spare communication links, each
  - Full duplex 4 lanes each direction@ 4Gbps signal rate
  - Equaling 2GB/s in each direction
  - Supports 5D torus topology

- Network Packets
  - 32B header, 0 to 512B data in 32B increments, 8B trailer
  - RDMA reads, writes, memory FIFO

- In NIC Collective operations
  - DP FltPt add, max, min
  - Integer add (signed/unsigned), max, min
  - Logical And, Or, Xor

Estimated Storage per Node

- Assume V vertices, H heavy vertices
- In, Out, Vis: 3V/8N bytes (1 bit per vertex)
- P: 8V/N bytes
- Index: 8*(V/64N) bytes (8 bytes per vertex)
- Edge list for 1 vertex: 264B on average
  - 8B vertex # + 32*8B for 32 edges
- InH, OutH, VisH: 3H/B bytes (1 bit per vertex)
  - Complete copy on each node
- PH: 8H (again complete copy per node)
- Edge list one 1 heavy vertex: 8B+4|Eh| (H at most 2^32)
- I/O buffers: 2*256*N

Total: \(272.5V/N + (16.4+8E_H)H + 512N\)
Storage/Node: Scale=35, N=2048

Only 16 GB available per Node

Highest GTEPS per Node

Storage/Node: Scale=41, N=98,304

Only 16 GB available per Node

Highest GTEPS per System
**BG/Q Network Bandwidth**

Saturation at 256B packets implies at most 36-50 (u,v,dir) messages per packet

Checconi and Petrini, “Traversing Trillions …”

---

**Traffic Due to Hybrid 1D Algorithm**

Huge reduction in messages;
This includes compression

Checconi and Petrini, “Traversing Trillions …”
**Observed Compression Effect**

- **Normal packing** is ~6B/edge
- **Compression**: using 7 bits/target vertex when packet holds many edges
- At levels with few edges, effect of header is larger.

<table>
<thead>
<tr>
<th>Green: my guess as to ave. edges per packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>~1.5</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

Checconi and Petrini, “Traversing Trillions …”

---

**Time/Level vs Graph Representation**

Checconi and Petrini, “Traversing Trillions …”
Effect of Multi-Threading Within Node

Constant scale 34 on 1024 nodes (1 rack)

These represent 2 and 4 threads respectively per core; worse than linear increase!

With 16 codes/node these numbers are better than linear!

Checoni and Petrini, “Traversing Trillions …”

Speedup Over Forward Algorithm

Different Graphs

Net speedup from 3X to 8X

a.k.a G500

scale 25

from Beamer, et al “Direction-Optimized…”
**Question**

- How do all these systems have only about 1 memory reference per TEP?
- Clearly they use the 30MB cache
- Also, I/O uses cache also
  - With set of atomics

**Observations on Memory**

- 16M vertices per node
  - Requires only 16M bits for each bit vector
  - Totaling $3 \times 16M/64 = 0.75$MB
- 2 256B I/O buffers for 2048 Nodes $\sim 1$MB
  - NICs can access cache directly
  - And perform atomic operations on them
- Together, these easily fit in cache
  - No memory references need for them
- System size growth to 100K nodes $\Rightarrow 50$MB of I/O
- P array too big for cache: 256MB
  - But each word written to at most once per vertex