The BFS Kernel: Applications and Implementations

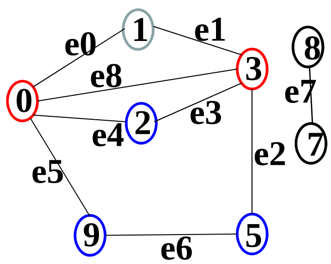
Peter M. Kogge
McCourtney Prof. of CSE
Univ. of Notre Dame
IBM Fellow (retired)

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
- From Wikipedia: https://en.wikipedia.org/wiki/Breadth-first_search
  - 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 1^{st} 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

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

```
while |In| != 0
    Out = {};
    for u in In do
        for v in some edge (u,v)
            if v not in Vis
                Out = Out U {v};
                Vis = Vis U {v};
                P[v] = u;
    In = Out;
```

Block executed for *each edge traversed*

TEPS = # of times/sec this block is executed

Sequential "Backward" BFS
Explore backwards from Untouched

```
while vertices were added in prior step
    Out = {};
    for v not in Vis do
        for u in some edge (u,v)
            if u in Vis
                Out = Out U {v};
                Vis = Vis U {v};
                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
- http://www.scottbeamer.net/pubs/beamer-sc2012.pdf

Edges Explored per Level

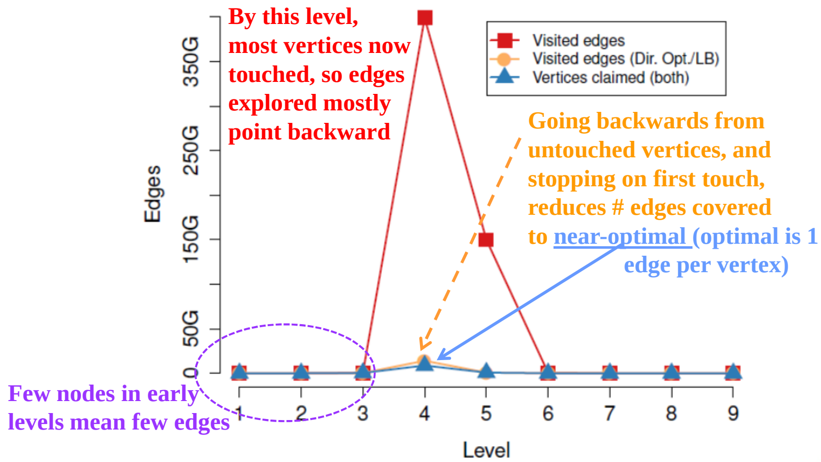


Fig. 5: Graph properties at each exploration level.

Checconi and Petrini, "Traversing Trillions …"

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

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

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

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 ≥1 endpoints
- **Rack**: Collection of blades
- **Domain**: set of cores that share same address space, all accessible via load/stores

2D Architectural Classification

System Interconnect

- **L**: Loosely coupled distributed memory
  - Commodity networking with software I/F
- **T**: Tightly coupled distributed memory
  - Specialized NICs & some H/W RDMA ops
- **S**: Shared Memory
  - Single domain in H/W
- **D**: Distributed Shared Memory
  - Single domain but S/W assist for remote references (typically via traps)

Core Architecture

- **H**: Heavyweight
- **L**: Lightweight
- **B**: BlueGene
- **X**: Multi-threaded
- **V**: Vector
- **O**: Other
- **G**: GPU-like
- **M**: a mix

Examples

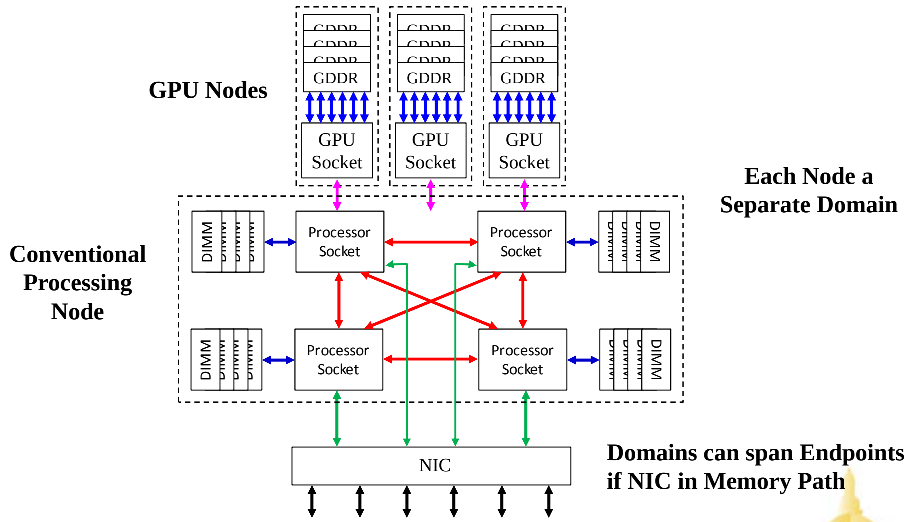
System Interconnect

- **T**: Tightly coupled
  - Cray systems with Aries NICs
- **L**: Loosely coupled
  - Infiniband Networking
- **S**: Shared Memory
  - SGI UV systems, XMT
- **D**: Dist. Shared Memory
  - Numascale

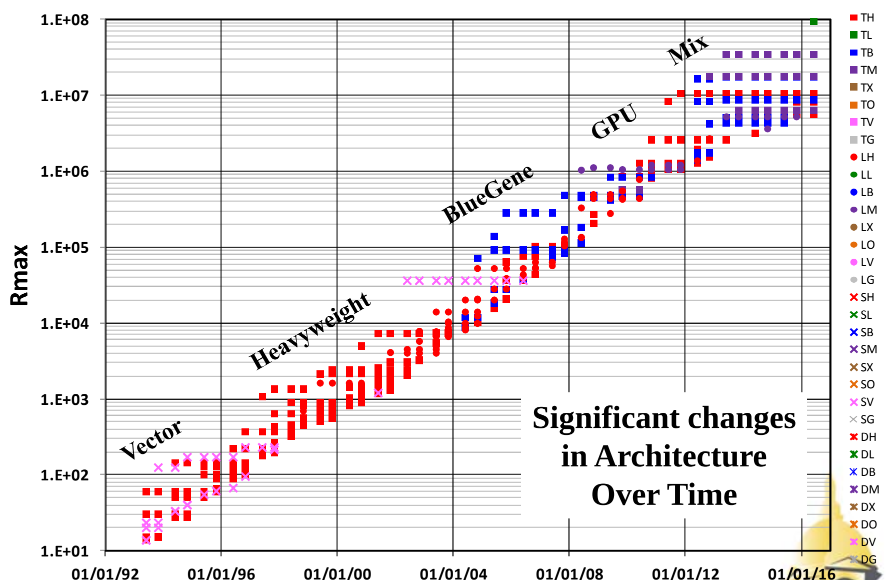
Core Architecture

- **H**: Heavyweight: Xeon
- **L**: Lightweight: ARM
- **B**: BlueGene
- **X**: Multi-threaded: XMT
- **V**: Vector: NEC SX
- **O**: Other: Convey
- **G**: GPU-like: Nvidia
- **M**: a mix

A Modern "Multi-Node" Endpoint



HPL Architectural Change

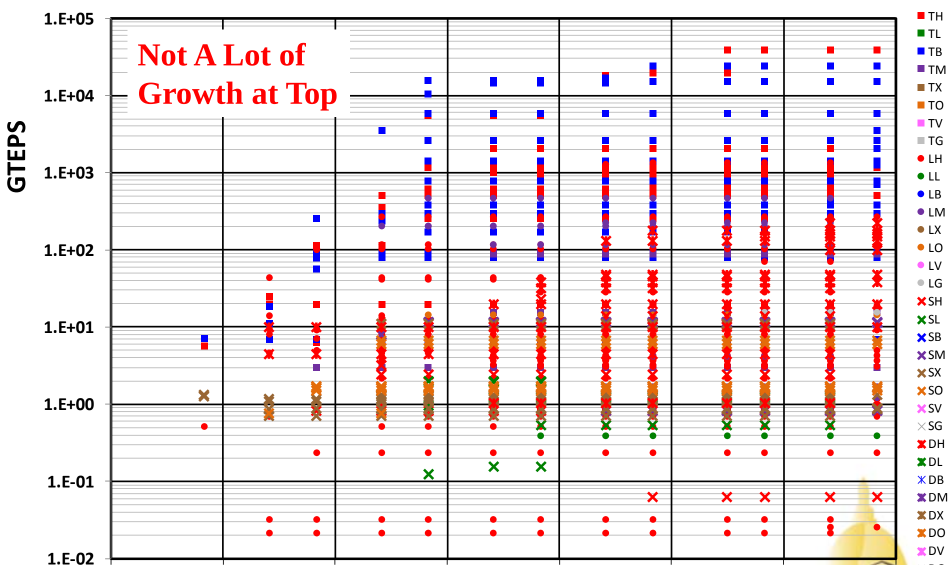


Key Architectural Parameters

- **R_{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**

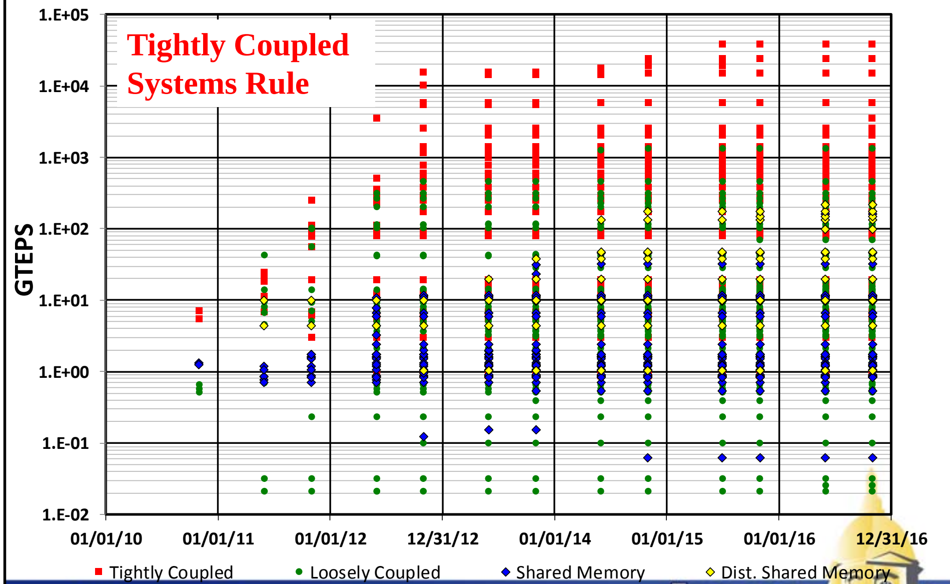
AND RATIOS

BFS Over Time

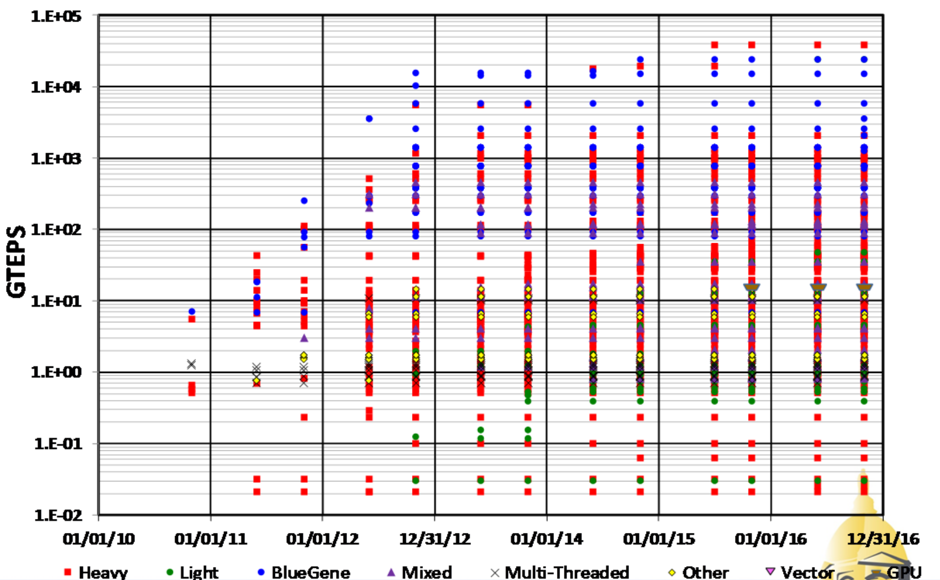


01/01/10 01/01/11 01/01/12 12/31/12 01/01/14 01/01/15 01/01/16 12/31/16

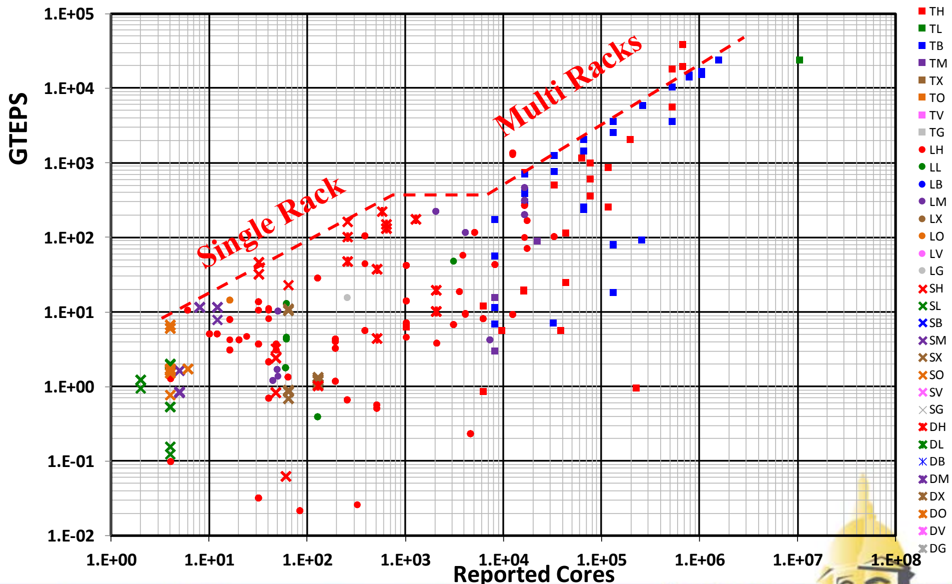
Trends by System Class



Trends by Core Architecture

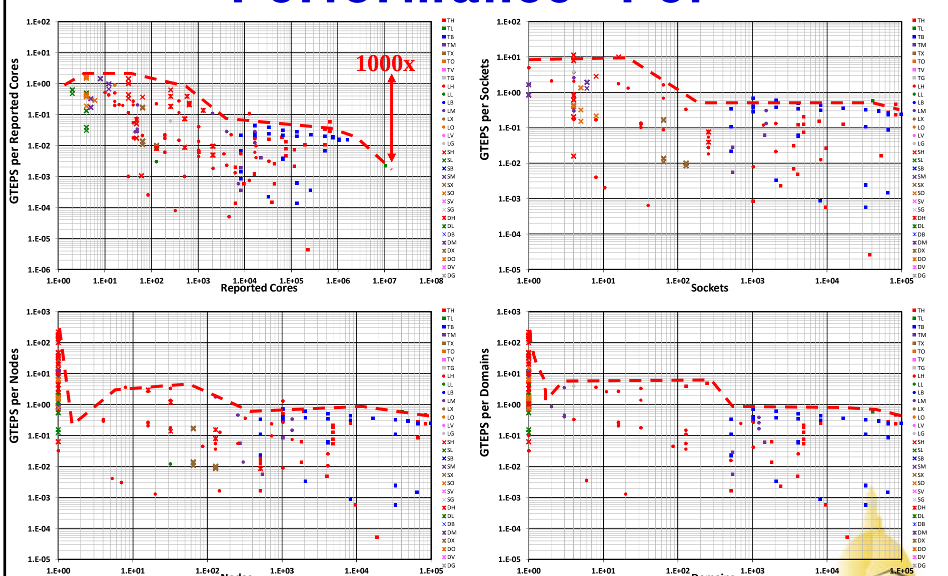


Performance vs # Cores

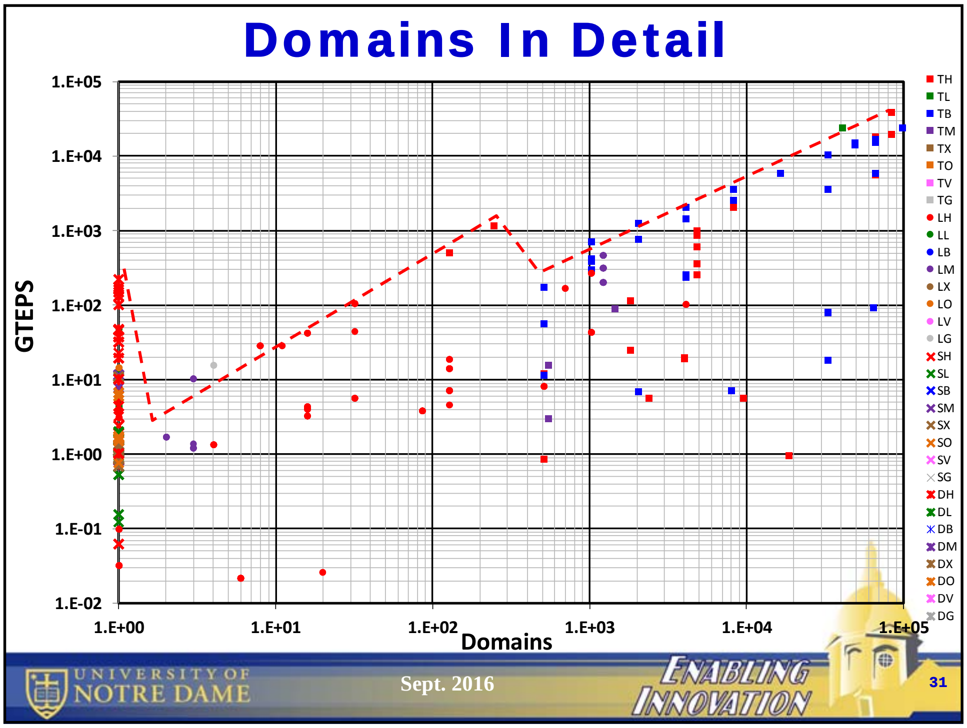


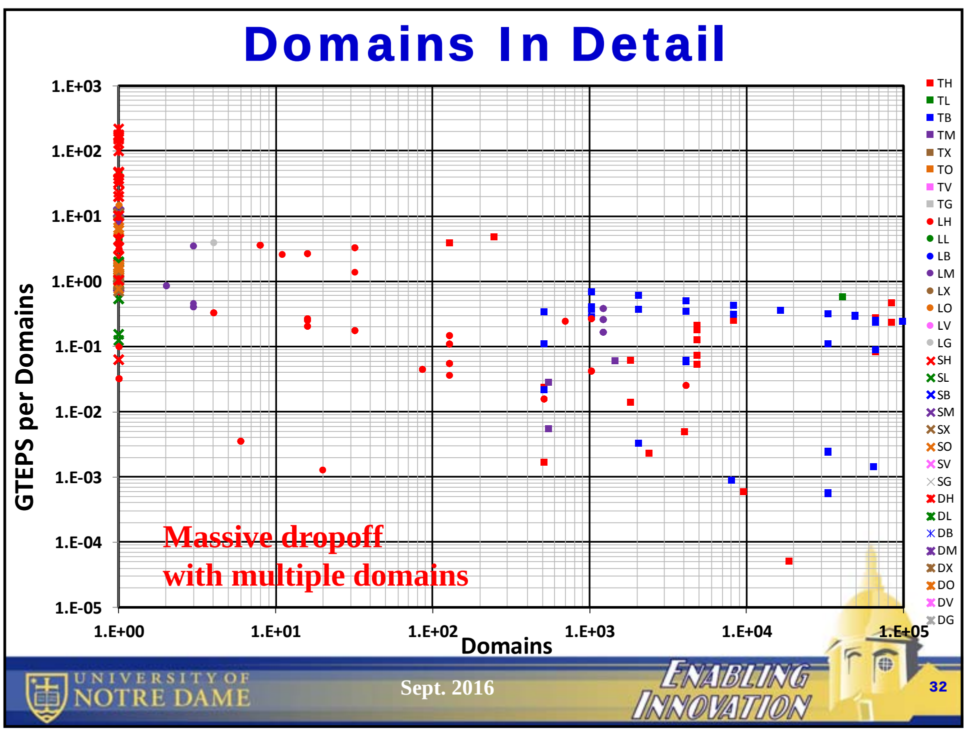
UNIVERSITY OF NOTRE DAME | Sept. 2016 | ENABLING INNOVATION | 29

Performance “Per”

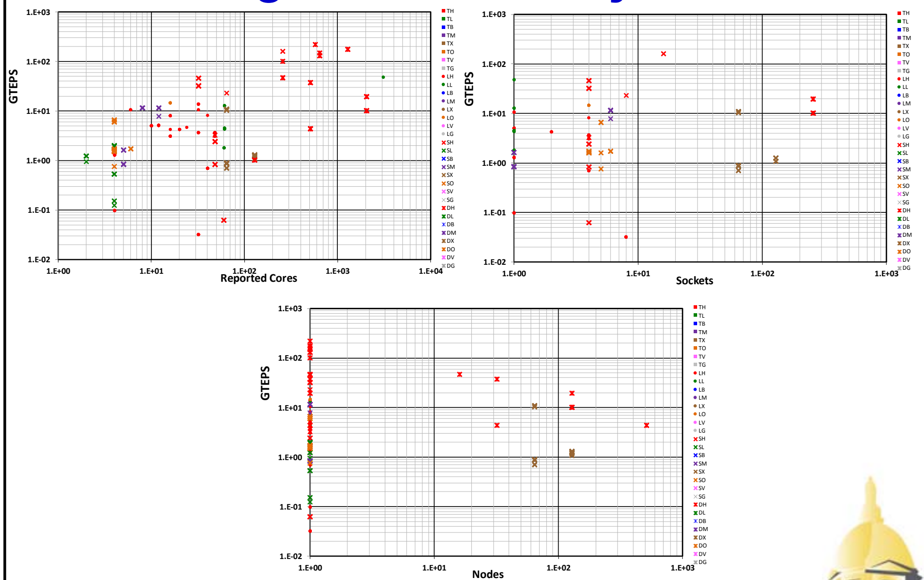


UNIVERSITY OF NOTRE DAME | Sept. 2016 | ENABLING INNOVATION | 30

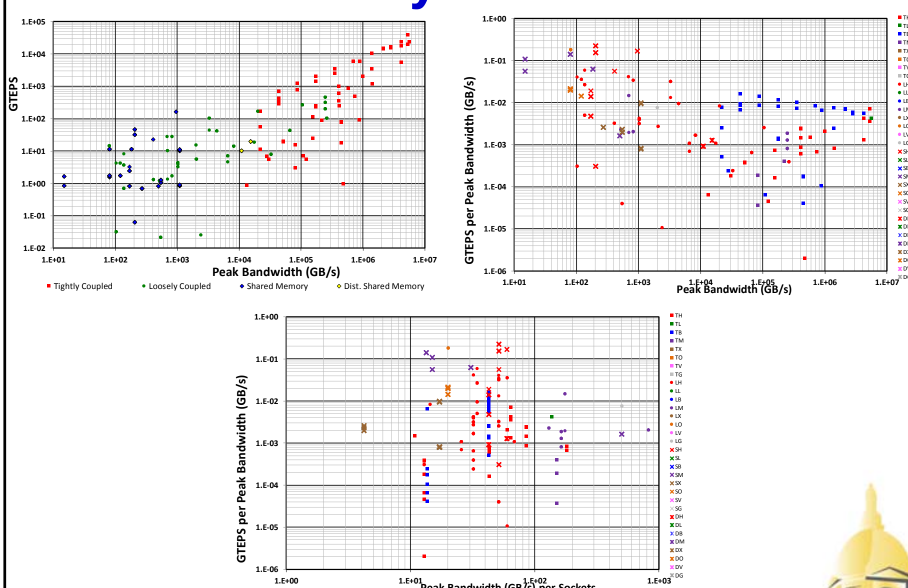

Domains In Detail
GTEPS
Domains
1.E+05
1.E+04
1.E+03
1.E+02
1.E+01
1.E+00
1.E-01
1.E-02
1.E+00
1.E+01
1.E+02
1.E+03
1.E+04
1.E+05
TH
TL
TB
TM
TX
TO
TV
TG
LH
LL
LB
LM
LX
LO
LV
LG
SH
SL
SB
SM
SX
SO
SV
SG
DH
DL
DB
DM
DX
DO
DV
DG
UNIVERSITY OF NOTRE DAME
Sept. 2016
ENABLING INNOVATION
31

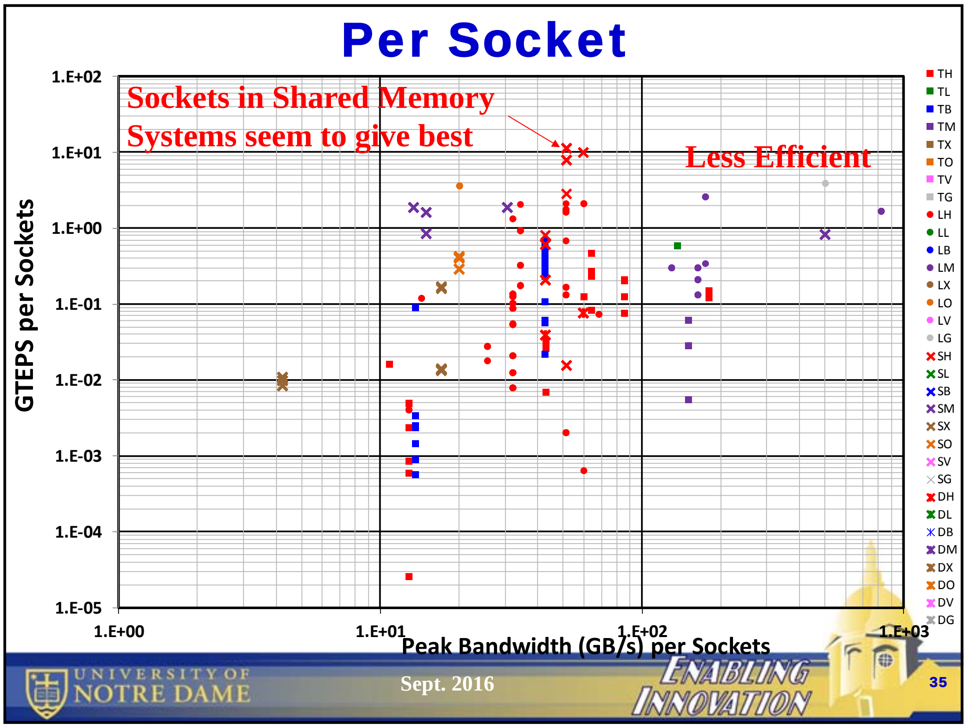

Domains In Detail
GTEPS per Domains
Domains
1.E+03
1.E+02
1.E+01
1.E+00
1.E-01
1.E-02
1.E-03
1.E-04
1.E-05
1.E+00
1.E+01
1.E+02
1.E+03
1.E+04
1.E+05
Massive dropoff
with multiple domains
TH
TL
TB
TM
TX
TO
TV
TG
LH
LL
LB
LM
LX
LO
LV
LG
SH
SL
SB
SM
SX
SO
SV
SG
DH
DL
DB
DM
DX
DO
DV
DG
UNIVERSITY OF NOTRE DAME
Sept. 2016
ENABLING INNOVATION
32

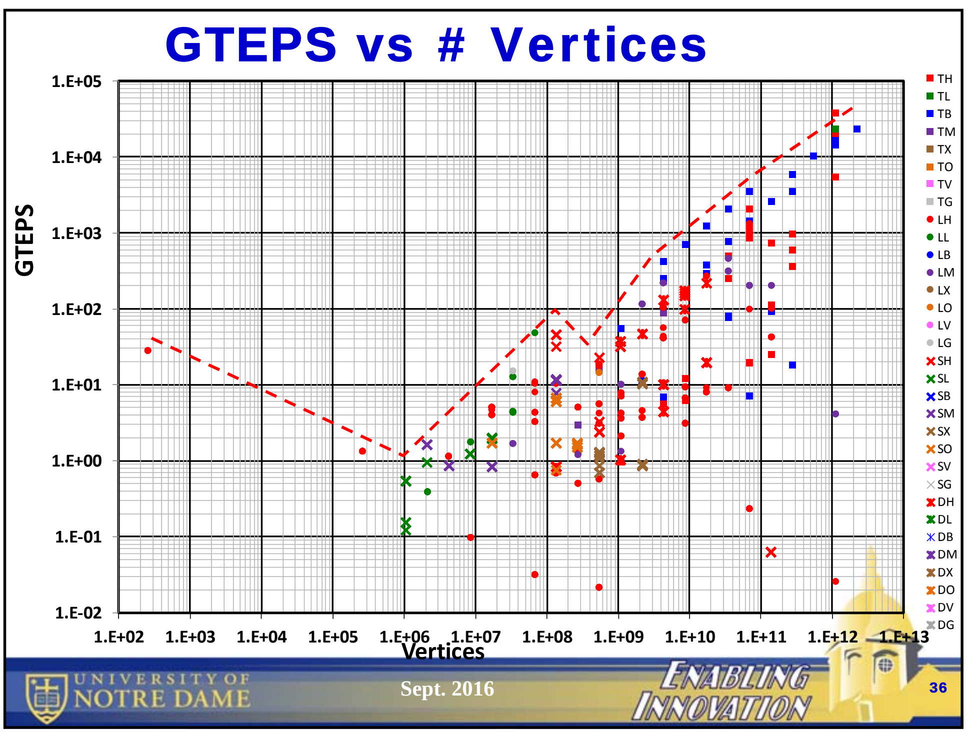
Single Domain Systems



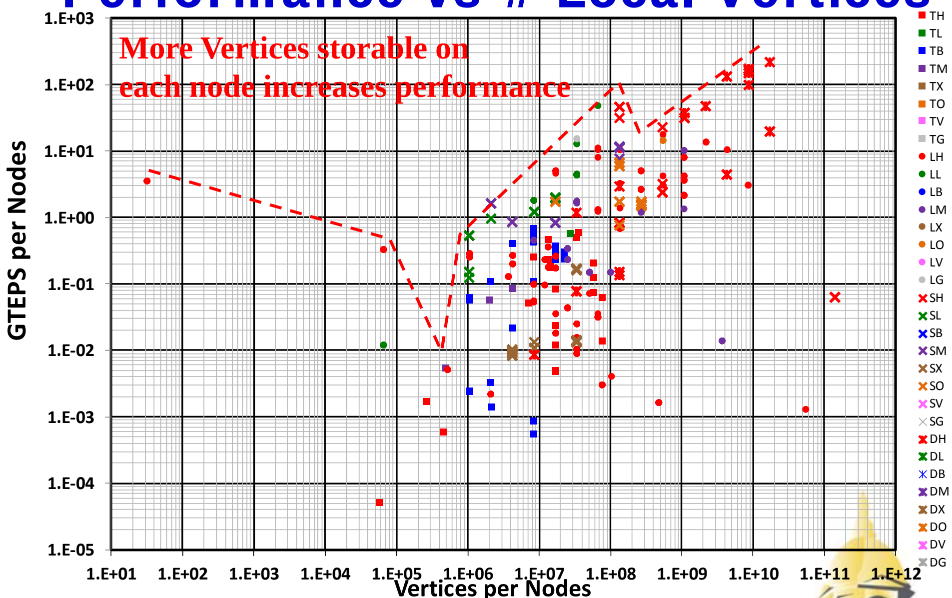
Memory Bandwidth



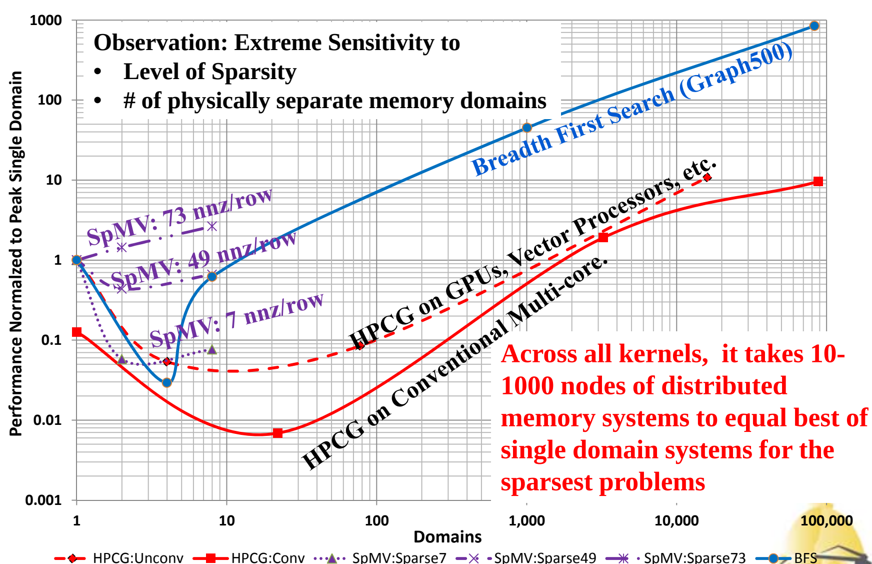

Per Socket
Sockets in Shared Memory
Systems seem to give best
Less Efficient
GTEPS per Sockets
1.E+02
1.E+01
1.E+00
1.E-01
1.E-02
1.E-03
1.E-04
1.E-05
1.E+00
1.E+01
1.E+02
1.E+03
Peak Bandwidth (GB/s) per Sockets
TH TL TB TM TX TO TV TG LH LL LB LM LX LO LV LG SH SL SB SM SX SO SV SG DH DL DB DM DX DO DV DG
UNIVERSITY OF NOTRE DAME
Sept. 2016
ENABLING INNOVATION
35


GTEPS vs # Vertices
GTEPS
1.E+05
1.E+04
1.E+03
1.E+02
1.E+01
1.E+00
1.E-01
1.E-02
1.E+02
1.E+03
1.E+04
1.E+05
1.E+06
1.E+07
1.E+08
1.E+09
1.E+10
1.E+11
1.E+12
1.E+13
Vertices
TH TL TB TM TX TO TV TG LH LL LB LM LX LO LV LG SH SL SB SM SX SO SV SG DH DL DB DM DX DO DV DG
UNIVERSITY OF NOTRE DAME
Sept. 2016
ENABLING INNOVATION
36

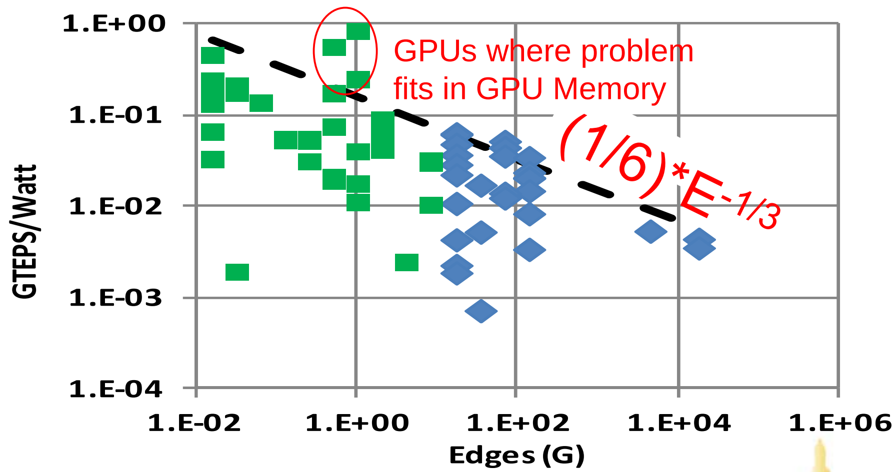
Performance vs # Local Vertices



Sparsity & Parallelism



TEPS / Watt



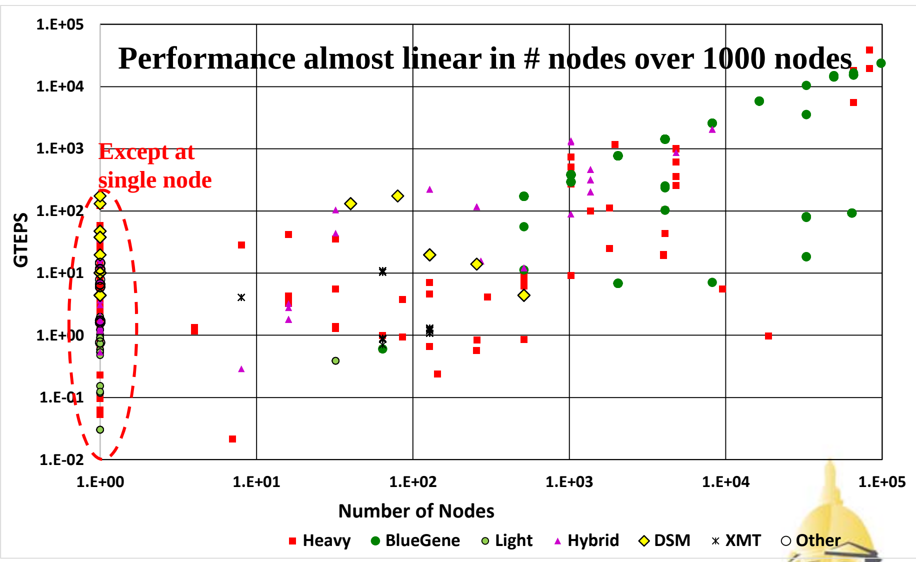
Green = Single Node, Blue = multi-node

Conclusions

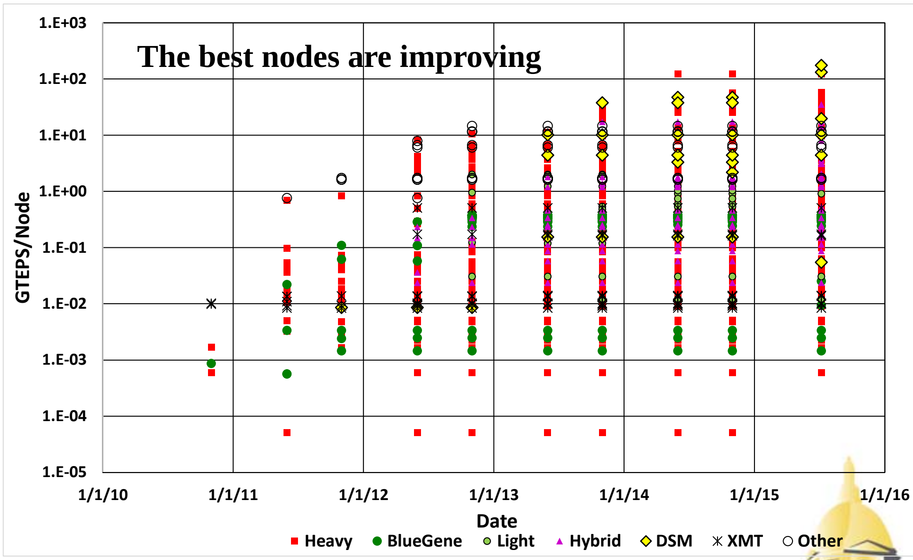
- 3 Performance regions
  - Single Domain: highest performance per core, … – by far
  - < 1 Rack
    - Significant drop-off from single domain
    - But excellent weak scaling
    - Especially shared memory vector machines
  - >1 Rack
    - Another drop-off from single rack
    - But again good scaling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

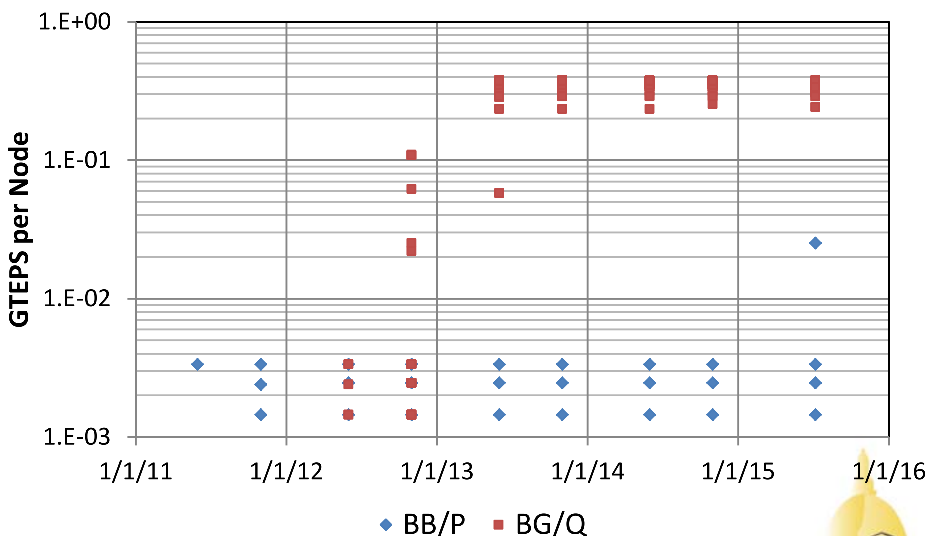


GTEPS / Node vs Time: All Systems



UNIVERSITY OF NOTRE DAME Sept. 2016 ENABLING INNOVATION 43

GTEPS / Node: BlueGene Only



UNIVERSITY OF NOTRE DAME Sept. 2016 ENABLING INNOVATION 44

Recent BG/Q Measurements

Date	Scale	GTEPS	Number of Nodes	Memory (GB)	GTEPS/ Node	Vertices	Vertices/ node	Cache Bits/ Vertex	Mem. Bytes/ Vertex	Memory B/W /TEP	Accesses per TEP
11/1/2014	33	172	512	8,192	3.36E-01	8.6E+09	1.68E+07	16.00	1024	127	0.99
11/1/2014	34	294	1024	16,384	2.87E-01	1.7E+10	1.68E+07	16.00	1024	148	1.16
11/1/2014	34	382	1024	16,384	3.73E-01	1.7E+10	1.68E+07	16.00	1024	114	0.89
11/1/2014	35	769	2048	32,768	3.75E-01	3.4E+10	1.68E+07	16.00	1024	114	0.88
7/8/2015	36	0.601	64	1024	9.40E-03	6.9E+10	1.07E+09	0.25	16	4541	35.34
11/1/2014	36	1427	4096	65,536	3.48E-01	6.9E+10	1.68E+07	16.00	1024	122	0.95
11/1/2014	37	2567	8192	131,072	3.13E-01	1.4E+11	1.68E+07	16.00	1024	136	1.06
11/1/2014	38	5848	16384	262,144	3.57E-01	2.7E+11	1.68E+07	16.00	1024	120	0.93
11/1/2014	40	14982	49152	786,432	3.05E-01	1.1E+12	2.24E+07	12.00	768	140	1.09
11/1/2014	41	23751	98304	1,572,860	2.42E-01	2.2E+12	2.24E+07	12.00	768	177	1.37

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)

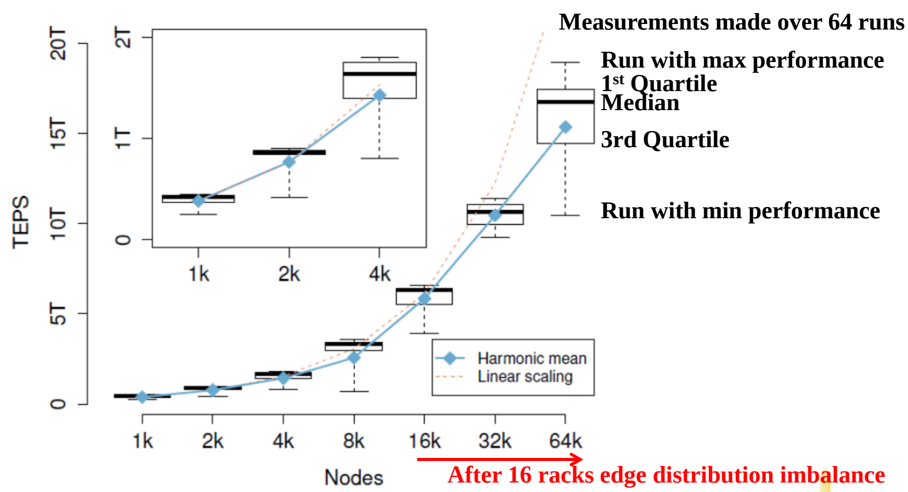


Fig. 10: Weak scaling.

After 16 racks edge distribution imbalance increases, causing reduced scaling

1K nodes = 1 rack

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

Date	Scale	GTEPS	Number of Nodes	Memory (GB)	GTEPS/ Node	Vertices	Vertices/ node	Cache Bits/ Vertex	Mem. Bytes/ Vertex	Memory B/W / TEP	Accesses per TEP
11/1/2014	33	172	512	8,192	3.36E-01	8.6E+09	1.68E+07	16.00	1024	127	0.99
11/1/2014	34	294	1024	16,384	2.87E-01	1.7E+10	1.68E+07	16.00	1024	148	1.16
11/1/2014	34	382	1024	16,384	3.73E-01	1.7E+10	1.68E+07	16.00	1024	114	0.89
11/1/2014	35	769	2048	32,768	3.75E-01	3.4E+10	1.68E+07	16.00	1024	114	0.88
7/8/2015	36	0.601	64	1024	9.40E-03	6.9E+10	1.07E+09	0.25	16	4541	35.34
11/1/2014	36	1427	4096	65,536	3.48E-01	6.9E+10	1.68E+07	16.00	1024	122	0.95
11/1/2014	37	2567	8192	131,072	3.13E-01	1.4E+11	1.68E+07	16.00	1024	136	1.06
11/1/2014	38	5848	16384	262,144	3.57E-01	2.7E+11	1.68E+07	16.00	1024	120	0.93
11/1/2014	40	14982	49152	786,432	3.05E-01	1.1E+12	2.24E+07	12.00	768	140	1.09
11/1/2014	41	23751	98304	1,572,860	2.42E-01	2.2E+12	2.24E+07	12.00	768	177	1.37

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

BG/Q Parallel BFS

```
while In != {} do
   dir = CalculateDirection;
   if dir = FORWARD
      for u in In_n do
         for v such that (u,v) in E do
            send(u, v, FORWARD) to owner(v);
   else
      for v not in In_n 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_n
         Vis_n = Vis_n U {v};
         Out_n = Out_n U {v};
         P[v] = u;
      else if u in In_n
         send(u, v, FORWARD) to owner(v);
```

Forward Step for non-heavies

Backward Step for non-heavies

Add v to frontier if it is not already touched

Forward Step for Heavies

```
OutH_n = {};
for u in InH do
   for each v from (u, v) in E_n do
      if v in H then
         VisH_n = VisH_n U {v};
         OutH_n, = OutH_n U {v};
         PH_n[v] = u;
      else
         Vis_n = Vis_n U {v};
         Out_n, = Out_n U {v};
         P_n[v] = u;
allreduce (VisH_n, OR);
allreduce (OutH_n, OR);
InH = OutH_n;
```

All nodes look at all heavies

But only process edges that are local

If target of edge is a heavy then update local copy of heavy data structures

If target of edge is not heavy then local node is owner of vertex's data, and update is again completely local

Need allreduce to combine all local copies of heavy data structures

Note! no messages needed in loop!!!

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 from (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};
             Out^H_n, = Out^H_n U {v};
             P^H_n[v] = u;
          else if u in In_n do
             Vis_n = Vis_n U {v};
             Out_n, = Out_n U {v};
             P_n[v] = u;
allreduce (Vis^H_n, OR);
allreduce (Out^H_n, OR);
In^H = Out^H_n;
```

If source of edge is heavy then update local copy of heavy data structures

If source is not heavy but local, then update local copy of non-heavy data structures. non-local non-heavy source handled by other loop

Need allreduce to combine all local copies of heavy data structures

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
- In^H, Out^H, Vis^H: 3H/B bytes (1 bit per vertex)
  - Complete copy on each node
- P^H: 8H (again complete copy per node)
- Edge list one 1 heavy vertex: $8B+4|E_h|$ (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.

1

1

~1.5

5

Many

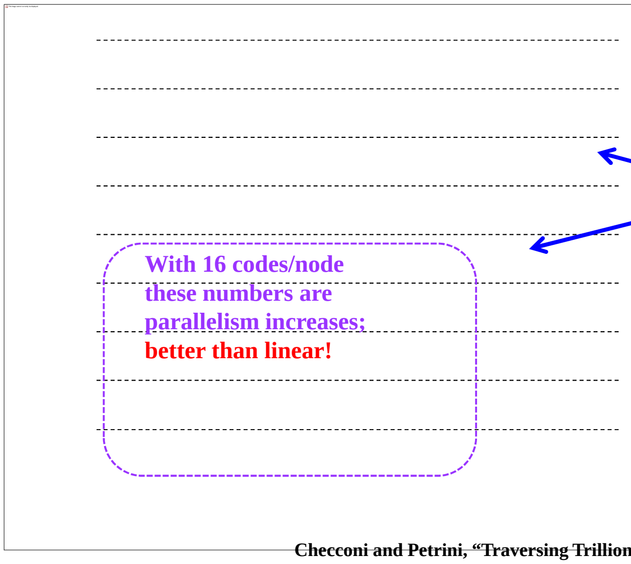
Green: my guess as to ave. edges per packet

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!

Checconi and Petrini, “Traversing Trillions …”

Speedup Over Forward Algorithm

Different Graphs

a.k.a G500 scale 25

from Beamer, et al “Direction-Optimized…”

Net speedup from 3X to 8X

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*16M/64 = 0.75MB
- 2 256B I/O buffers for 2048 Nodes ~1MB
  - 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 => 50MB of I/O
- P array too big for cache: 256MB
  - But each word written to at most once per vertex