Breadth First Search on APEnet+

Enrico Mastrostefano[1] Massimo Bernaschi[2] Mauro Bisson[2]
Davide Rossetti (for the APEnet coll.)[3]

[1]Sapienza Università di Roma
[2]Istituto per le Applicazioni del Calcolo, IAC-CNR, Rome, Italy
[3]APE group, INFN Roma, Italy

IA^3 Workshop on Irregular Applications: Architectures & Algorithms
– SC12 – Nov. 11 2012

Outline

1. Workload: distributed BFS on large graphs
2. Platform: 3D Torus interconnect, support for GPU peer-to-peer
3. Results: on 4-8 nodes with APEnet+

Rationale

From the Workshop web site:

- *Many data intensive scientific applications are by nature irregular. . . Current supercomputing systems are organized around components optimized for data locality and regular computation. Developing irregular applications on them demands a substantial effort, and often leads to poor performance.*

Rationale

From the Workshop web site:

- *Many data intensive scientific applications are by nature irregular. . . Current supercomputing systems are organized around components optimized for data locality and regular computation. Developing irregular applications on them demands a substantial effort, and often leads to poor performance.*
- *The solutions needed to address these challenges can only come by considering the problem from all perspectives: from micro- to system-architectures. . . from algorithm design to data characteristics.*

Rationale

From the Workshop web site:

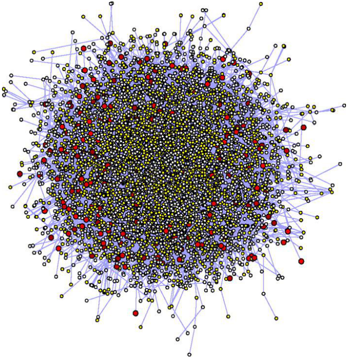
- *Many data intensive scientific applications are by nature irregular. . . Current supercomputing systems are organized around components optimized for data locality and regular computation. Developing irregular applications on them demands a substantial effort, and often leads to poor performance.*
- *The solutions needed to address these challenges can only come by considering the problem from all perspectives: from micro- to system-architectures. . . from algorithm design to data characteristics.*
- *Only collaborative efforts among researchers with different expertise, including end users, domain experts, and computer scientists, could lead to significant breakthroughs.*

Rationale

From the Workshop web site:

- *Many data intensive scientific applications are by nature irregular... Current supercomputing systems are organized around components optimized for data locality and regular computation. Developing irregular applications on them demands a substantial effort, and often leads to poor performance.*
- *The solutions needed to address these challenges can only come by considering the problem from all perspectives: from micro- to system-architectures... from algorithm design to data characteristics.*
- *Only collaborative efforts among researchers with different expertise, including end users, domain experts, and computer scientists, could lead to significant breakthroughs.*
- We (clearly :) match them all, You'll see ...

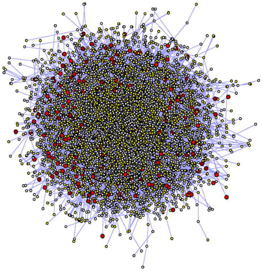
Large Graphs



- Large scale networks are often represented as large graphs with having up to billions of edges
- Power-law degree distribution

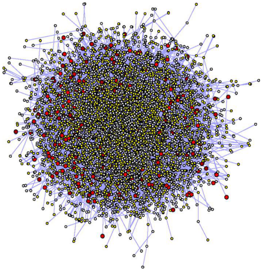
High performance graph algorithms

- Most of graph algorithms have low arithmetic intensity and irregular memory access patterns



High performance graph algorithms

- Most of graph algorithms have low arithmetic intensity and irregular memory access patterns
- How do modern architectures perform running such algorithms?

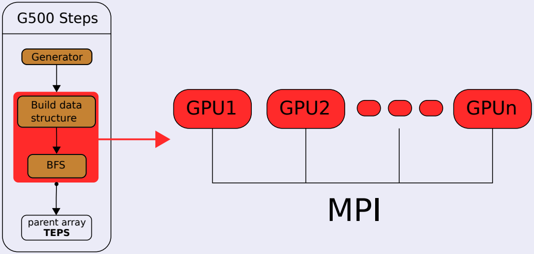


High performance graph algorithms

- Most of graph algorithms have low arithmetic intensity and irregular memory access patterns
- How do modern architectures perform running such algorithms?
- Several graph-theoretical challenges: DIMACS9, SCA#2, Graph 500

Overview

- Distributed Breadth First Search (BFS)
- Implementation for GPU clusters
- Programming paradigm: CUDA + MPI
- Developed according to the Graph 500 specifications.
  Performance metric: Traversed Edges Per Second (TEPS)



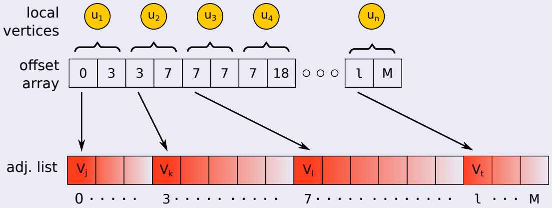
Distributed data structure

Edge list

- Edge list with: $< V >= 2^{SCALE}$; $< M >= 16 * 2^{SCALE}$
- Each task generates a subset of the edge list in the form: $(U_0, V_0), (U_1, V_1), \ldots$
- Edges are assigned to processes via a simple rule:
  edge $(U_i, V_j) \in P_k$ if $U_i \mod \#P == k$

Compressed Sparse Row (CSR) data structure

- CSR is simple and has minimal memory requirements



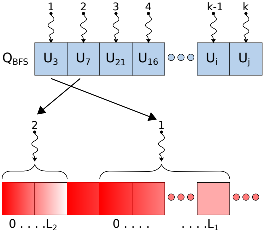
Straightforward implementation of BFS on a cluster of GPUs

Data mapping

- Each vertex U_i of Q_{BFS} is assigned to one thread t_i
- Each thread t_i visits all the neighbors V_j of its vertex

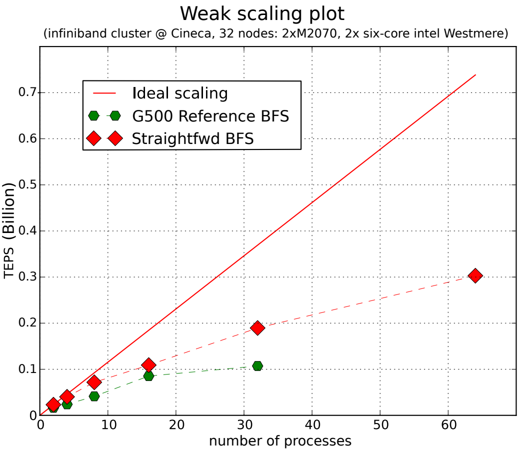
GPU-related issues

- Threads workloads are unbalanced
- Memory access patterns can be irregular



Algorithm rely on atomic (add) operations.

Straightforward BFS: Results

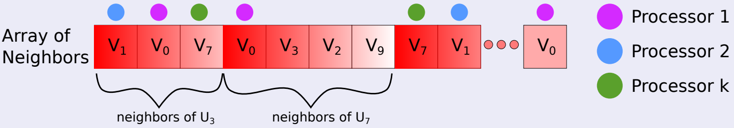


Poor TEPS scaling

Straightforward BFS: Issues

Communication-related issues

- Multiple copies of the same vertex are sent



Issues and Solutions

We used one thread for each vertex in the queue. $|Q| = k$ ....

.... $|Neighbors| = m$. We want to use as many threads as the number of neighbors

Neighbors of vertices in the queue are not-contiguous in the Adjacency list array...

...We want a contiguous array of neighbors

We send/recv multiple copies...

...We want to prune the array that we send

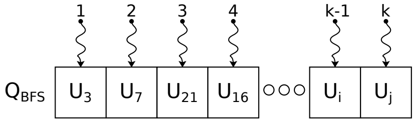
Beyond the straightforward BFS: Sort-Unique BFS

1. Build an array of offsets and compute the total number of neighbors, say m
2. Start m threads, map threads to neighbors and build a contiguous array of neighbors
3. With m threads prune the contiguous array of neighbors
4. Exchange vertices with other processes and update the parent array

Sort-Unique BFS

Recipe #1: build the new offset and compute the total number of neighbors

- Start k threads, each element of Q_{BFS} is assigned to one thread



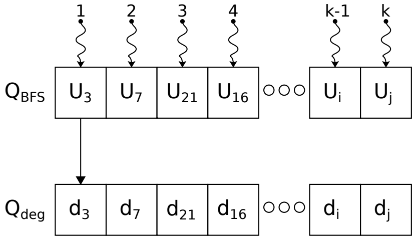
- Build Q_{deg}, by substituting each vertex with its degree

- Perform a **prefix-sum** operation on Q_{deg} to build the **New Offset** array (by using the Thrust library)

Sort-Unique BFS

Recipe #1: build the new offset and compute the total number of neighbors

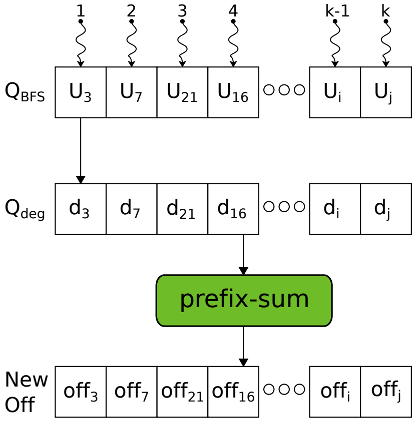
- Start k threads, each element of Q_{BFS} is assigned to one thread
- Build Q_{deg}, by substituting each vertex with its degree
- Perform a **prefix-sum** operation on Q_{deg} to build the **New Offset** array (by using the Thrust library)



Sort-Unique BFS

Recipe #1: build the new offset and compute the total number of neighbors

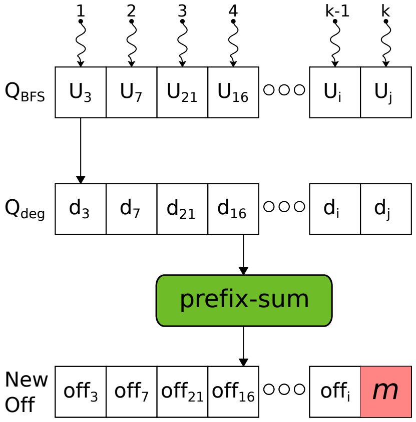
- Start k threads, each element of Q_{BFS} is assigned to one thread
- Build Q_{deg}, by substituting each vertex with its degree
- Perform a **prefix-sum** operation on Q_{deg} to build the **New Offset** array (by using the Thrust library)



Sort-Unique BFS

Recipe #1: build the new offset and compute the total number of neighbors

- Start k threads, each element of Q_{BFS} is assigned to one thread
- Build Q_{deg}, by substituting each vertex with its degree
- Perform a **prefix-sum** operation on Q_{deg} to build the **New Offset** array (by using the Thrust library)

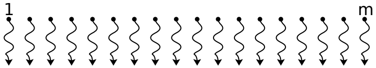


The last element of **New Offset** is: $m = \sum_{i \in Q_{BFS}} d_i$

Sort-Unique BFS

Recipe #2: map threads to neighbors and build a contiguos array of neighbors

- Start *m* threads

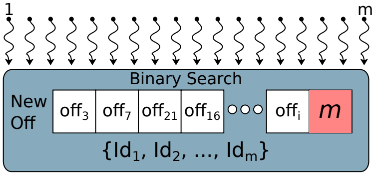


- Each thread performs a **binary search** on **New Offset** and finds its index
- Each thread reads from the Adj list the element corresponding to the index
- and write it in the **Next Level Frontier Set** (NLFS).

Sort-Unique BFS

Recipe #2: map threads to neighbors and build a contiguos array of neighbors

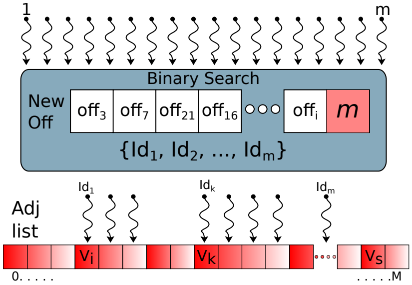
- Start *m* threads
- Each thread performs a **binary search** on **New Offset** and finds its index
- Each thread reads from the Adj list the element corresponding to the index
- and write it in the **Next Level Frontier Set** (NLFS).



Sort-Unique BFS

Recipe #2: map threads to neighbors and build a contiguos array of neighbors

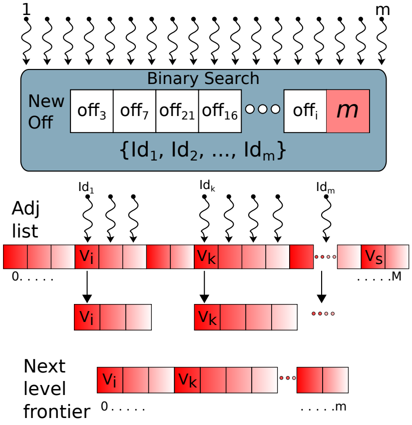
- Start *m* threads
- Each thread performs a **binary search** on **New Offset** and finds its index
- Each thread reads from the Adj list the element corresponding to the index
- and write it in the **Next Level Frontier Set** (NLFS).



Sort-Unique BFS

Recipe #2: map threads to neighbors and build a contiguos array of neighbors

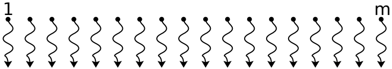
- Start *m* threads
- Each thread performs a **binary search** on **New Offset** and finds its index
- Each thread reads from the Adj list the element corresponding to the index
- and write it in the Next Level **Frontier Set** (NLFS).



Sort-Unique BFS

Recipe #3: prune the Next Level Frontier Set

- Start *m* threads



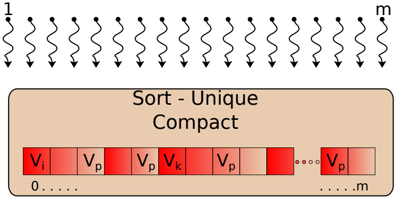
- Perform a **sort-unique** operation on the **Next Level Frontier Set** (by using the Thrust library)

- and compact it to *n* unique elements

Sort-Unique BFS

Recipe #3: prune the Next Level Frontier Set

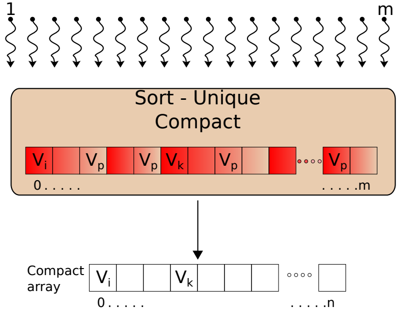
- Start *m* threads
- Perform a **sort-unique** operation on the **Next Level Frontier Set** (by using the Thrust library)
- and compact it to *n* unique elements



Sort-Unique BFS

Recipe #3: prune the Next Level Frontier Set

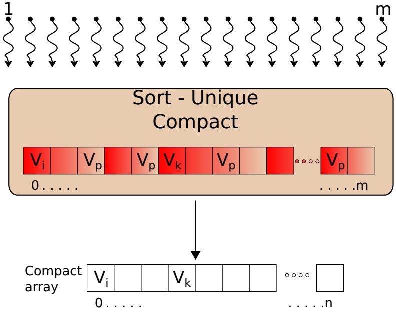
- Start *m* threads
- Perform a **sort-unique** operation on the **Next Level Frontier Set** (by using the Thrust library)
- and compact it to *n* unique elements



Sort-Unique BFS

Recipe #3: prune the Next Level Frontier Set

- Start m threads
- Perform a **sort-unique** operation on the **Next Level Frontier Set** (by using the Thrust library)
- and compact it to n unique elements

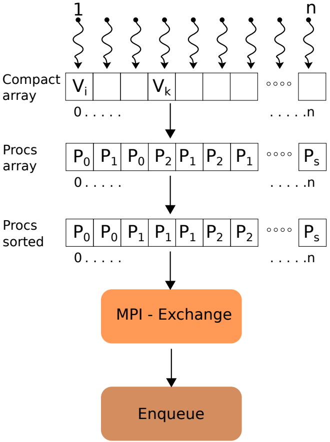


- Unique ratio $\frac{m}{n} \sim 20$

Sort-Unique BFS: communication and enqueue

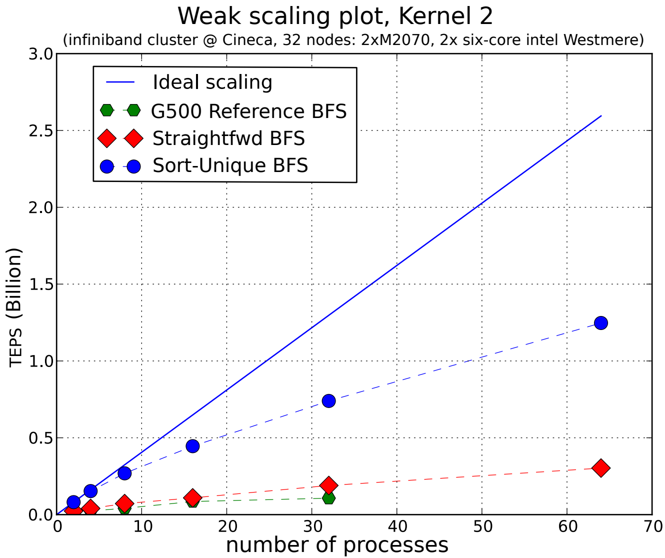
Recipe #4: Exchange vertices and update the parent array

- Start *n* threads
- Substitute vertices with tasks
- Sort by process id (by using the Thrust library)
- Exchange non-local edges
- Update the parent array and Enqueue



- If $Q_{BFS} == 0$ quit.

Sort-Unique BFS: Results



Sort-Unique BFS: weak scaling analysis

- Time spent in computation is almost constant
- Time spent in communication increases with N_P
- Sort-Unique cuts $\simeq$ 90% of vertices

SCALE	N_P	kernels time	mpi time	NLFS	NLFS-after-SU
21	1	0.68	0.0	37,651,259	1,043,789
22	2	0.85	0.1	37,906,934	1,678,486
23	4	0.85	0.4	37,739,872	2,688,755
24	8	0.85	0.5	58,416,610	4,502,903
25	16	0.9	0.6	45,334,918	5,519,616
26	32	0.95	0.7	58,863,642	8,703,456
27	64	1.01	0.9	42,174,869	9,316,248

NOTE: figures are for third-level of BFS.

The Graph 500 List

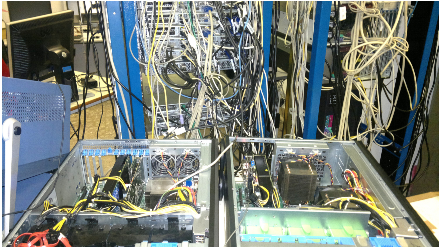
November 2011 | June 2011 | November 2010

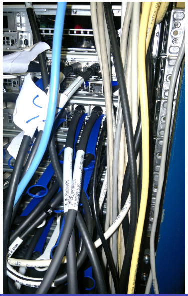
Complete Results - November 2011

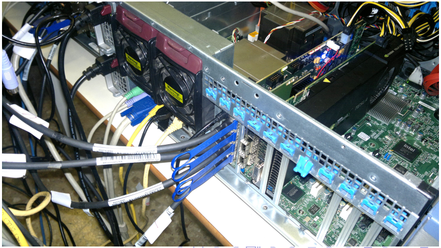
Rank	Machine	Owner	Problem Size	TEPS	Implementation
1	NNSA/SC Blue Gene/Q Prototype II (4096 nodes / 65,536 cores)	NNSA and IBM Research, T.J. Watson	32	254,349,000,000	Custom
2	Hopper (1800 nodes / 43,200 cores)	LBL	37	113,368,000,000	Custom
2	Lomonosov (4096 nodes / 32,768 cores)	Moscow State University	37	103,251,000,000	Custom
3	TSUBAME (2732 processors / 1366 nodes / 16,392 CPU cores)	GSIC Center, Tokyo Institute of Technology	36	100,366,000,000	Custom
18	Blacklight (512 processors)	PSC	32 (Small)	4,452,270,000	Custom
19	Todi (176 AMD Interlagos, 176 NVIDIA Tesla X2090)	CSCS	28	3,059,970,000	Custom GPU Result
20	Dingus (Convey HC-1ex - 1 node / 4 cores, 4 FPGAs)	SNL	28	1,758,682,718	Convey Custom
28	PLX (32 nodes each with 2 Tesla M2070)	CINECA, Bologna, Italy	26	1,357,320,000	GPU Customized

Platform: APEnet+ test setup

- 8-nodes setup
- dual-socket Westmere Xeon servers
- 2D Torus 4*x*2*x*1 topology
- one/two NVIDIA Fermi 2050 GPUs per node
- GPU ECC OFF







APEnet+ card

- FPGA based (Altera Stratix IV)
- 3D Torus, 6 bidirectional links up to 34 Gbps raw
- PCIe X8 Gen2 in X16 slot (peak BW 4+4 GB/s)
- Network Processor, off-loading engine integrated on FPGA
- Zero-copy RDMA host interface
- Direct GPU peer-to-peer logic
- Industry standard QSFP+ cabling (copper & optical)

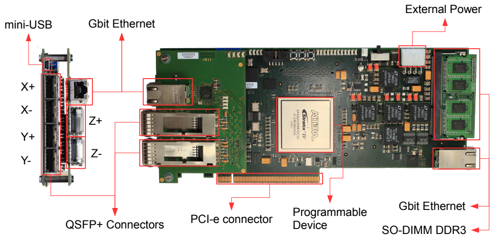


Figure : APEnet+ card, front view

Data exchange between GPUs

- (In general) GPUs cannot exchange data directly
- Data staging on host memory represents a bottleneck on multi-GPUs systems
- NVIDIA Fermi GPUs introduced HW support for *peer-to-peer* (P2P) over PCIexpress
- SW support present since CUDA 4.0

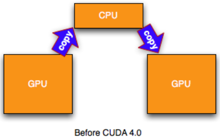


Figure : Data exchange between GPUs before CUDA 4.0

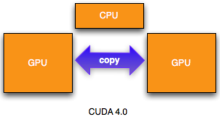


Figure : Direct memory copy between GPUs enabled by CUDA 4.0

Data exchange with 3rd party devices

- CUDA 4.1, unofficial P2P support for 3rd party devices
- APEnet+ is 1st (only?) non-NVIDIA device to support the P2P HW protocol, directly across PCIexpress
- CUDA 5.0, 3rd party access via BAR1 for Kepler

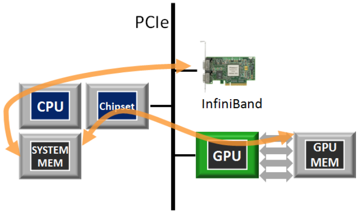


Figure : Standard interconnects, data staging on host memory

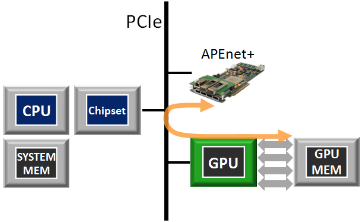


Figure : Direct P2P data transfer of GPUs data to/from APEnet+ across the PCIexpress bus

A possible confusion . . .

GPU-aware MPI, ever heard of them ?

- OSU MVAPICH2 and OpenMPI (SVN trunk)
- hide data staging on host memory, *i.e.* `MPI_Send` and `MPI_recv` accept GPU memory pointers.
- rely on NVIDIA UVA

Useful but not GPU *peer-to-peer* with interconnect

Results: BFS on APEnet+

- OpenMPI/IB using MPI_Send/Recv
- APEnet+ using native RDMA PUT (needs padding)

Traversed Edges Per Second, Strong Scaling, $|V| = 2^{20}$

NP	INFINIBAND	APENET
1	6.2×10^7	6.2×10^7
2	7.8×10^7	1.0×10^8
4	8.2×10^7	1.3×10^8
8	2.0×10^8	?

Traversed Edges Per Second, Weak Scaling, $|V| = 2^{SCALE}$

NP	SCALE	MPI/IB	APEnet+
1	19	5.6×10^7	6.0×10^7
2	20	7.9×10^7	1.0×10^8
4	21	1.1×10^8	1.5×10^8
8	22	2.7×10^8	?

APEnet+ vs. MPI/IB

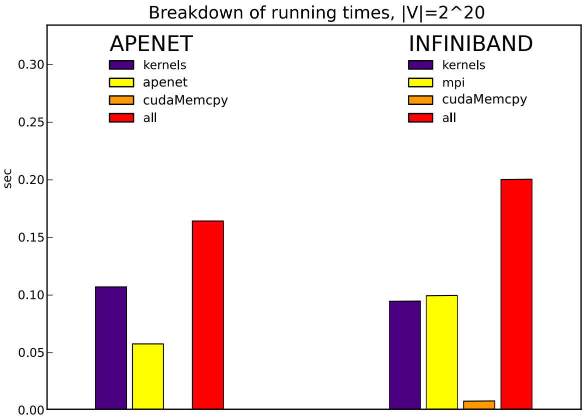


Figure : Execution time breakdown, $SCALE = 20$, $N_p = 4$, on one process among four.

Network pattern

- all-to-all communication
- msg size, rise and fall
- sharp peak at level 3

eg. for $N_p = 4$ and $SCALE = 20$:

level 1
src=0 dest=1 len=64
src=0 dest=2 len=64
src=0 dest=3 len=64

level 2
src=1 dest=0 len=588
src=2 dest=0 len=256
src=3 dest=0 len=268KB
src=0 dest=1 len=576
src=2 dest=1 len=192
src=3 dest=1 len=263KB
src=0 dest=2 len=576
src=1 dest=2 len=5.8KB
src=3 dest=2 len=261KB
src=0 dest=3 len=576
src=1 dest=3 len=5.9KB
src=2 dest=3 len=192

level 3
src=1 dest=0 len=1.0MB
src=2 dest=0 len=1.6MB
src=3 dest=0 len=1.6MB
src=0 dest=1 len=1.6MB
src=2 dest=1 len=1.6MB
src=3 dest=1 len=1.6MB
src=0 dest=2 len=1.6MB
src=1 dest=2 len=1.6MB
src=3 dest=2 len=1.6MB
src=0 dest=3 len=1.6MB
src=1 dest=3 len=1.6MB
src=2 dest=3 len=1.6MB

level 4
src=1 dest=0 len=1MB
src=2 dest=0 len=1MB
src=3 dest=0 len=1MB
src=0 dest=1 len=1MB
src=2 dest=1 len=1MB
src=3 dest=1 len=1MB
src=0 dest=2 len=1MB
src=1 dest=2 len=1MB
src=3 dest=2 len=1MB
src=0 dest=3 len=1MB
src=1 dest=3 len=1MB
src=2 dest=3 len=1MB

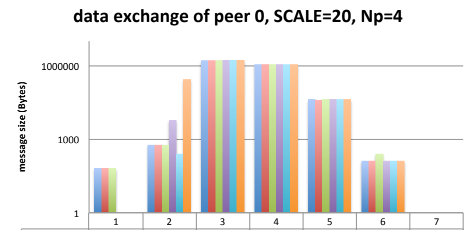
level 5
src=1 dest=0 len=128
src=3 dest=0 len=41KB
src=2 dest=0 len=41KB
src=0 dest=1 len=41KB
src=2 dest=1 len=41KB
src=3 dest=1 len=40KB
src=1 dest=2 len=41KB
src=0 dest=2 len=40KB
src=3 dest=2 len=42KB
src=1 dest=3 len=41KB
src=0 dest=3 len=41KB
src=2 dest=3 len=42KB

level 6
src=3 dest=0 len=128
src=2 dest=0 len=128
src=0 dest=1 len=128
src=2 dest=1 len=128
src=3 dest=1 len=128
src=1 dest=2 len=128
src=0 dest=2 len=128
src=3 dest=2 len=128
src=1 dest=3 len=192
src=0 dest=3 len=256
src=2 dest=3 len=128

Network pattern

- all-to-all communication
- msg size, rise and fall
- sharp peak at level 3

eg. for $N_p = 4$ and $SCALE = 20$:

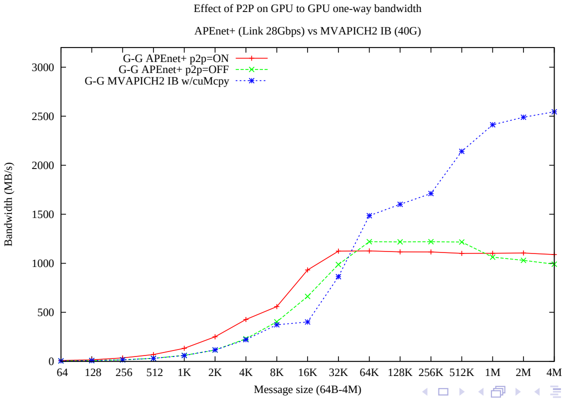


	1	2	3	4	5	6	7
to peer 1	64	576	1622336	1073792	41024	128	0
to peer 2	64	576	1628352	1077504	40192	128	0
to peer 3	64	576	1627072	1077440	41280	256	0
from peer 1	0	5888	1657664	1072896	41536	128	0
from peer 2	0	256	1643072	1080384	41408	128	0
from peer 3	0	268416	1647680	1075328	41728	128	0

understanding the performance difference

using basic network benchmarks as a guide:

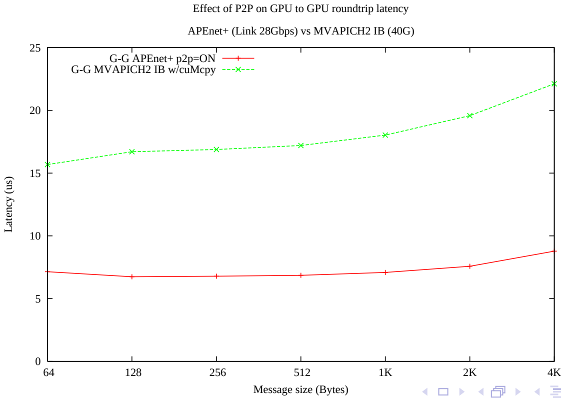
- 2-nodes uni-directional bandwidth test, GPU-to-GPU; p2p=OFF is APEnet+ staging in host memory



understanding the performance difference

using basic network benchmarks as a guide:

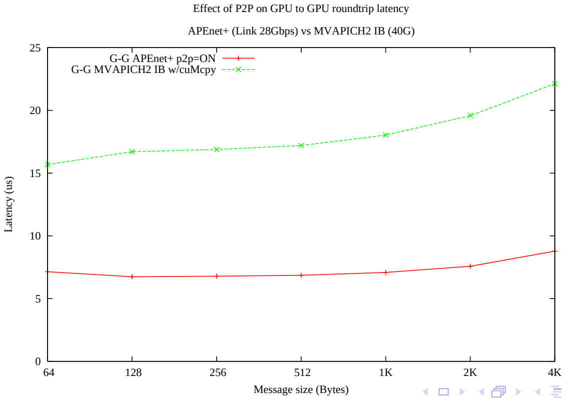
- 2-nodes uni-directional bandwidth test, GPU-to-GPU; p2p=OFF is APEnet+ staging in host memory
- APEnet+ round-trip latency with GPU peer-to-peer



understanding the performance difference

using basic network benchmarks as a guide:

- 2-nodes uni-directional bandwidth test, GPU-to-GPU; p2p=OFF is APEnet+ staging in host memory
- APEnet+ round-trip latency with GPU peer-to-peer
- the MVAPICH2 result on OSU MPI bandwidth test is for reference



Conclusions

Summary

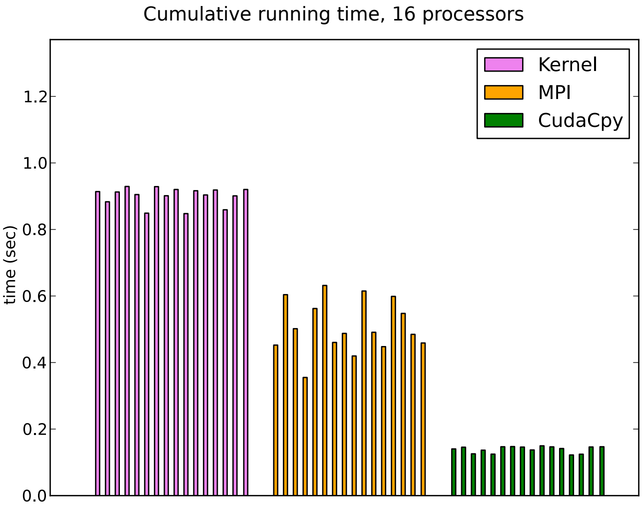
- Distributed BFS on multi-GPUs that relies on pruning
- Good scaling properties
- Up to 3 billions TEPS with 128 GPUs (19 rank in graph500)
- APEnet+ 1st attempt at GPU peer-to-peer

Future Work

- CUDA streams to overlap computation with communication
- P2P among GPUs on the *same* host
- APEnet+ is reconfigurable, space for HW optimizations

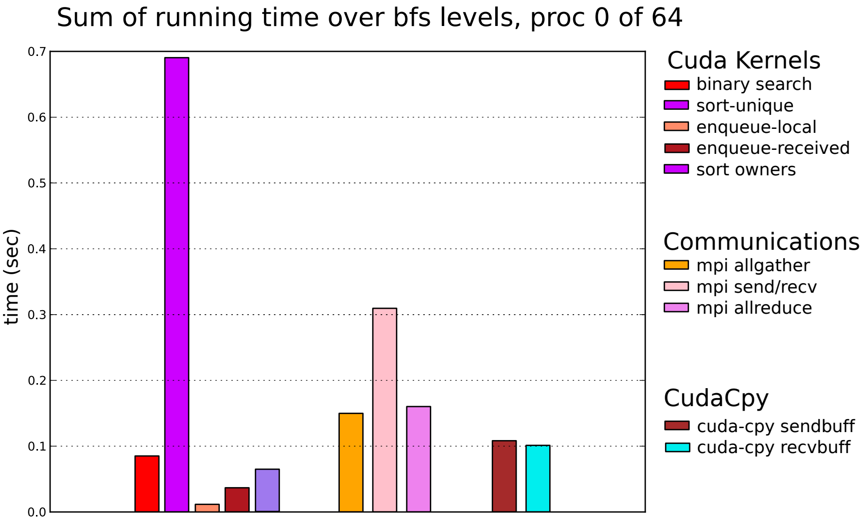
Backup slides

K2: balancing

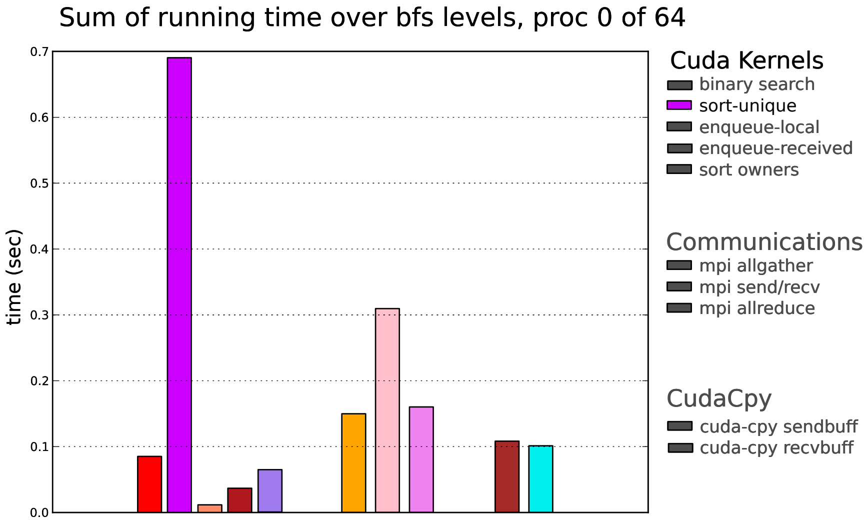


Computations and communications among processes are well balanced

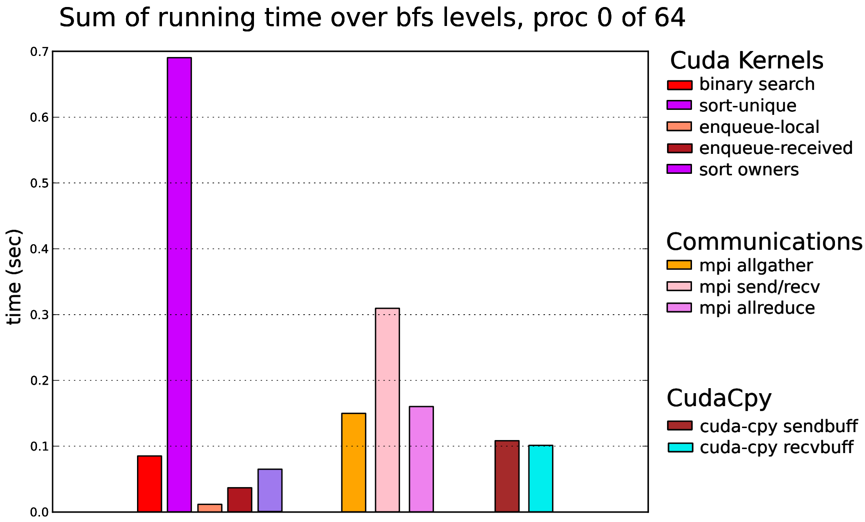
K2: cuda kernels times



K2: cuda kernels times



K2: cuda kernels times



References I

- Roberto Ammendola, Andrea Biagioni, Ottorino Frezza, Francesca Lo Cicero, Alessandro Lonardo, Pier Paolucci, Roberto Petronzio, Davide Rossetti, Andrea Salamon, Gaetano Salina, Francesco Simula, Nazario Tantalo, Laura Tosoratto, and Piero Vicini, *Apenet+: a 3d toroidal network enabling petaflops scale lattice qcd simulations on commodity clusters*.

- Aydin Buluc and Kamesh Madduri, *Parallel breadth-first search on distributed memory systems*.

- Deepayan Chakrabarti, Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos, *R-mat: A recursive model for graph mining*, IN SDM (2004).

- Andrew Grimshaw Duane Merrill, Michael Garland, *High performance and scalable gpu graph traversal*, Tech. report, Nvidia, 2011.

References II

- Andy Yoo et al., *A scalable distributed parallel breadth-first search algorithm on bluegene/l*, Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference, 2005.
- Pawan Harish and P. J. Narayanan, *Accelerating large graph algorithms on the gpu using cuda*, 2007, pp. 197–208.
- Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani, *Kronecker graphs: An approach to modeling networks*, J. Mach. Learn. Res. **11** (2010), 985–1042.
- Huiwei Lv, Guangming Tan, Mingyu Chen, and Ninghui Sun, *Understanding parallelism in graph traversal on multi-core clusters*, Computer Science - Research and Development (2012), 1–9.

References III

- Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (eds.), *Accelerating cuda graph algorithms at maximum warp*, 2011.
- Brian W. Barrett James A. Ang Richard C. Murphy, Kyle B. Wheeler, *Introducing the graph 500*, 2010.
- Koji Ueno and Toyotaro Suzumura, *Highly scalable graph search for the graph500 benchmark*, Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing (New York, NY, USA), HPDC '12, ACM, 2012, pp. 149–160.