Large Graph on multi-GPUs

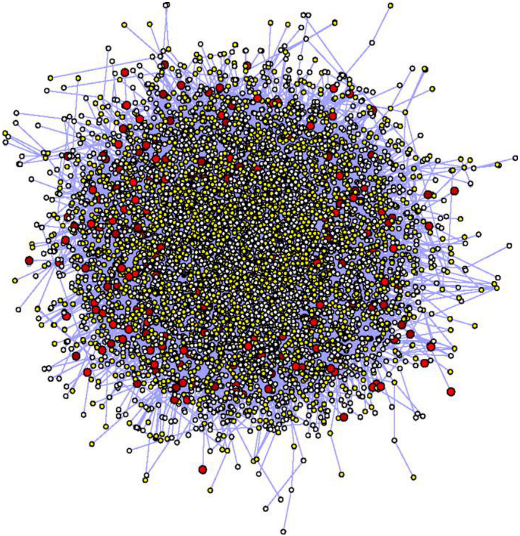
Enrico Mastrostefano[1] Massimo Bernaschi[2] Massimiliano Fatica[3]

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

May 11, 2012

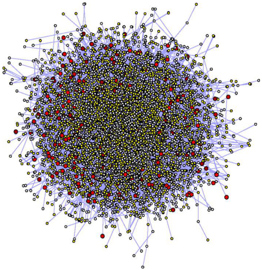
Outline

Large Graphs



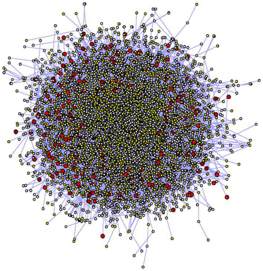
Large Graphs

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



Large Graphs

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



Large Graphs

- Most of graph algorithms have low arithmetic intensity and irregular memory access patterns
- How do modern architectures perform running such algorithms?
- Large-scale benchmark for data-intensive application

Large Graphs

- Most of graph algorithms have low arithmetic intensity and irregular memory access patterns
- How do modern architectures perform running such algorithms?
- Large-scale benchmark for data-intensive application
- “This is the first serious approach to complement the Top 500 with data-intensive applications ...” (from www.graph500.org)

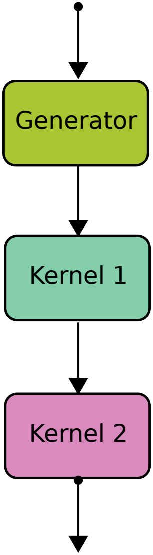
Large Graphs

- Most of graph algorithms have low arithmetic intensity and irregular memory access patterns
- How do modern architectures perform running such algorithms?
- Large-scale benchmark for data-intensive application
- "This is the first serious approach to complement the Top 500 with data-intensive applications ..." (from www.graph500.org)
- The core of the benchmark is a set of graph algorithms

graph500: specifications

Generator

- Generate the edge list with real-world properties (RMAT generator).
- minimum size: 2^{28} vertices and $16 * 2^{28}$ edges ($268,435,456$ vertices and $8,589,934,592$ edges)



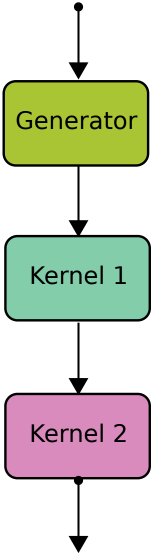
graph500: specifications

Generator

- Generate the edge list with real-world properties (RMAT generator).
- minimum size: 2^{28} vertices and $16 * 2^{28}$ edges ($268,435,456$ vertices and $8,589,934,592$ edges)

Kernel 1

- Build the data structure that represent the graph: timed!
- Data structure cannot be modified by subsequent kernels



graph500: specifications

Generator

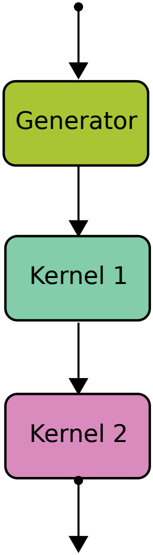
- Generate the edge list with real-world properties (RMAT generator).
- minimum size: 2^{28} vertices and $16 * 2^{28}$ edges ($268, 435, 456$ vertices and $8, 589, 934, 592$ edges)

Kernel 1

- Build the data structure that represent the graph: timed!
- Data structure cannot be modified by subsequent kernels

Kernel 2

- Perform a Breadth First Search (BFS) visit starting from a random vertex: timed!
- Output the parent array.



graph500: specifications

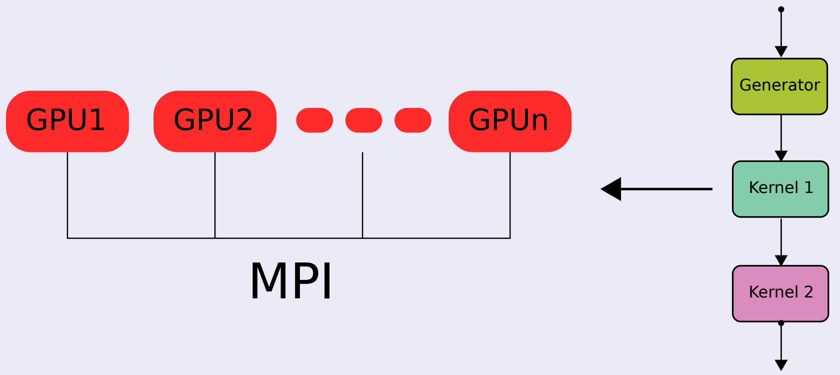
Outputs

- Execution time of K1
- Execution time of K2,
- TEPS: Traversed Edges Per Second in the BFS (actually the ranking is based only on TEPS)

Outline

graph500 on multi-GPUs

CUDA + MPI



Generate the edge list

Edge list

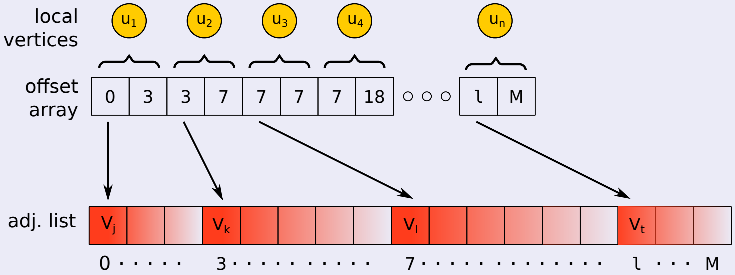
- We have to generate: $|V| = 2^{SCALE}$; $|M| = 16 * 2^{SCALE}$
- Each task generates a subset of the edge list in the form: $(U_0, V_0), (U_1, V_1), \ldots$
- Edges are assigned to processes via a simple rule: edge $(U_i, V_j) \in P_k$ if $U_i \mod \#P == k$

Build the data structure

Data structure

- We transform the edge list in a Compressed Sparse Row (CSR) data structure
- CSR is simple and has minimal memory requirements

Compressed Sparse Row (CSR)

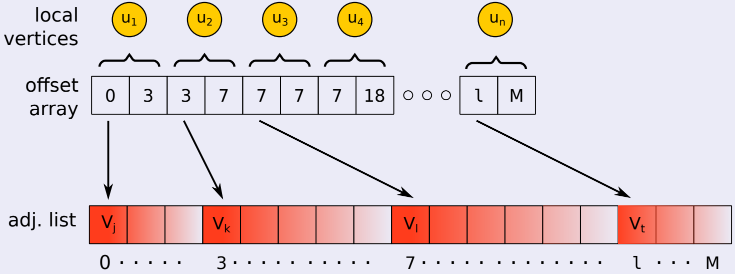


Build the data structure

Data structure

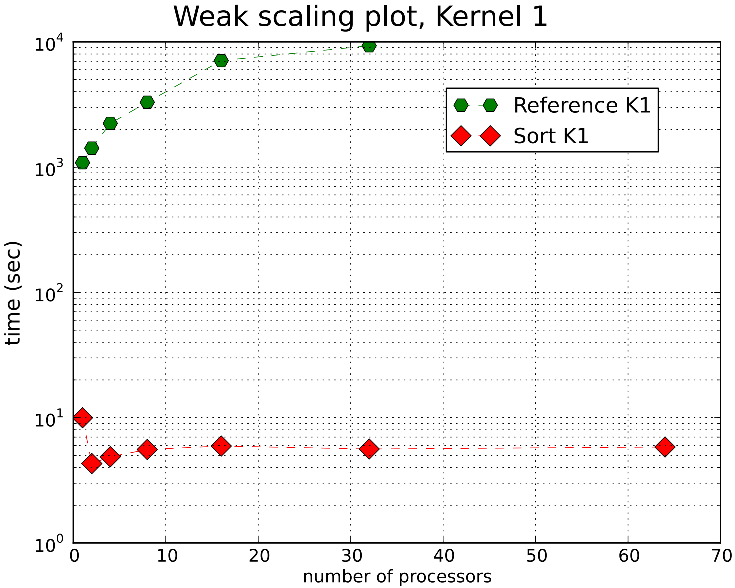
- We transform the edge list in a Compressed Sparse Row (CSR) data structure
- CSR is simple and has minimal memory requirements

Compressed Sparse Row (CSR)



The core of the algorithm is a sort

K1 results



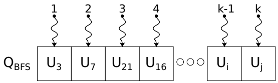
Weak scaling plot, Kernel 1
time (sec)
number of processors
Reference K1
Sort K1
10^4
10^3
10^2
10^1
10^0
0 10 20 30 40 50 60 70

Outline

Straightforward BFS

Queue based BFS

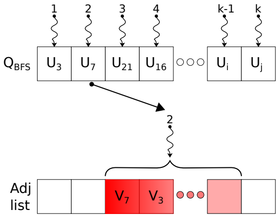
- Each vertex U_i of Q_{BFS} is assigned to one thread t_i
- Each thread t_i visits all the neighboring V_j of its vertex
- If V_j is local: visit it
- if V_j is not local: send to its owner
- receive vertices V_k from other processes



Straightforward BFS

Queue based BFS

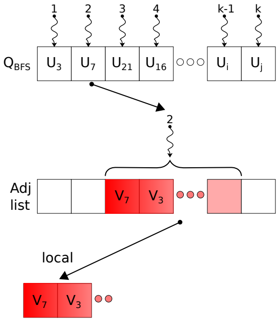
- Each vertex U_i of Q_{BFS} is assigned to one thread t_i
- Each thread t_i visits all the neighboring V_j of its vertex
- If V_j is local: visit it
- if V_j is not local: send to its owner
- receive vertices V_k from other processes



Straightforward BFS

Queue based BFS

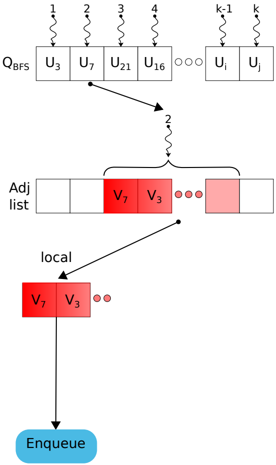
- Each vertex U_i of Q_{BFS} is assigned to one thread t_i
- Each thread t_i visits all the neighboring V_j of its vertex
- If V_j is local: visit it
- if V_j is not local: send to its owner
- receive vertices V_k from other processes



Straightforward BFS

Queue based BFS

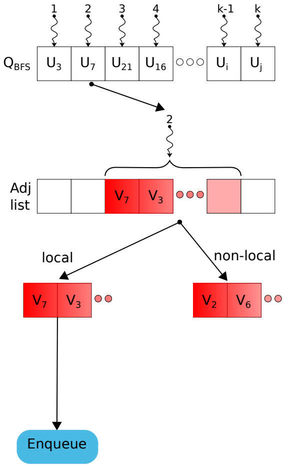
- Each vertex U_i of Q_{BFS} is assigned to one thread t_i
- Each thread t_i visits all the neighboring V_j of its vertex
- If V_j is local: visit it
  - update the predecessor array
  - enqueue V_j
- if V_j is not local: send to its owner
- receive vertices V_k from other processes



Straightforward BFS

Queue based BFS

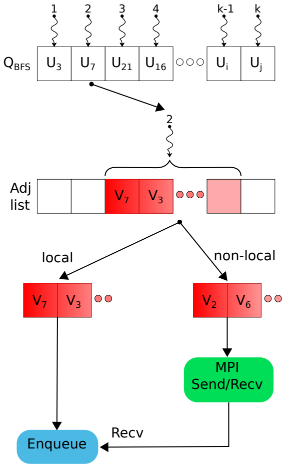
- Each vertex U_i of Q_{BFS} is assigned to one thread t_i
- Each thread t_i visits all the neighboring V_j of its vertex
- If V_j is local: visit it
  - update the predecessor array
  - enqueue V_j
- if V_j is not local: send to its owner
- receive vertices V_k from other processes



Straightforward BFS

Queue based BFS

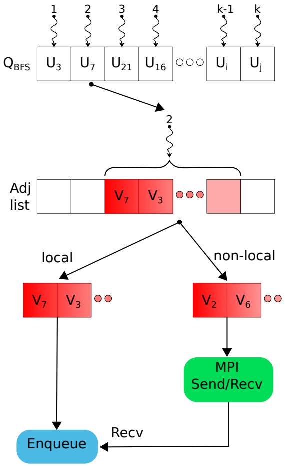
- Each vertex U_i of Q_{BFS} is assigned to one thread t_i
- Each thread t_i visits all the neighboring V_j of its vertex
- If V_j is local: visit it
  - update the predecessor array
  - enqueue V_j
- if V_j is not local: send to its owner
- receive vertices V_k from other processes
  - update the predecessor array
  - enqueue V_k



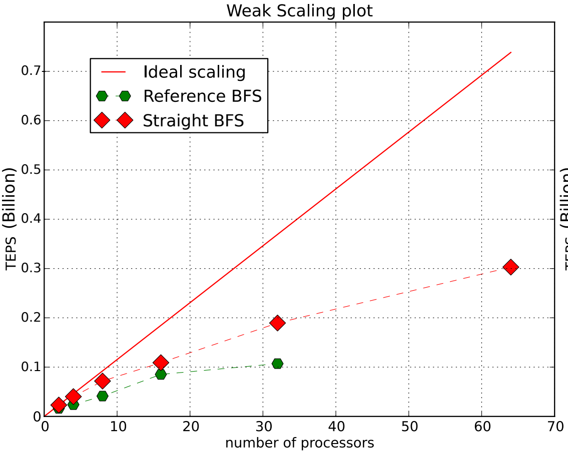
Straightforward BFS

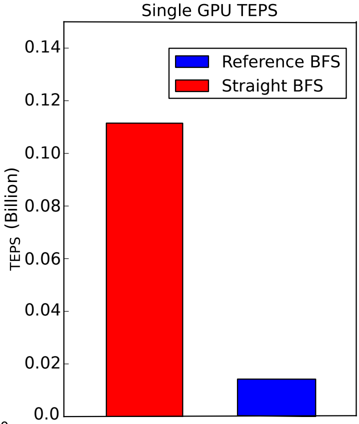
Queue based BFS

- Each vertex U_i of Q_{BFS} is assigned to one thread t_i
- Each thread t_i visits all the neighboring V_j of its vertex
- If V_j is local: visit it
  - update the predecessor array
  - enqueue V_j
- if V_j is not local: send to its owner
- receive vertices V_k from other processes
  - update the predecessor array
  - enqueue V_k
- Output parent array and TEPS



Straightforward BFS: Results

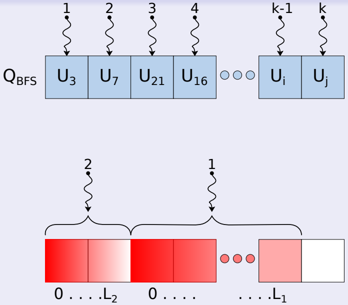




Straightforward BFS: Issues

Gpu-related issues

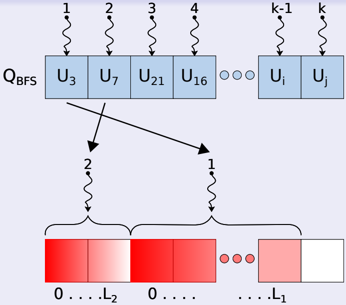
- Threads workloads are unbalanced when threads visit different adjacency lists



Straightforward BFS: Issues

Gpu-related issues

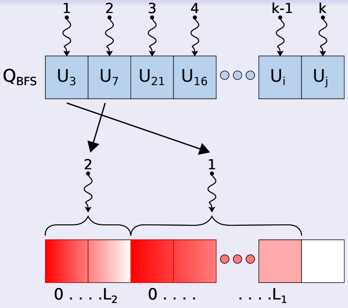
- Threads workloads are unbalanced when threads visit different adjacency lists
- Memory access patterns can be irregular and threads that belong to the same warp may access non-contiguos regions of memory



Straightforward BFS: Issues

Gpu-related issues

- Threads workloads are unbalanced when threads visit different adjacency lists
- Memory access patterns can be irregular and threads that belong to the same warp may access non-contiguos regions of memory

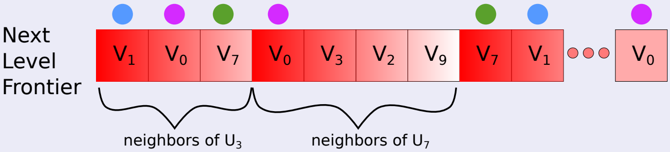


Algorithms rely on the use of **Atomic Add**

Straightforward BFS: Issues

MPI-related issues

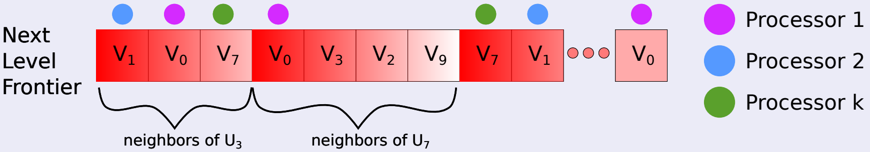
- We don't process non-local vertices so the array to send contains multiple copies of the same vertices.
- Multiple copies of the same vertex are sent to the owner.



Straightforward BFS: Issues

MPI-related issues

- We don't process non-local vertices so the array to send contains multiple copies of the same vertices.
- Multiple copies of the same vertex are sent to the owner.



Outline

Beyond the straightforward BFS

Beyond the straightforward BFS

- We use k threads to visit neighbors
- We want to use as many threads as the number of neighbors

Beyond the straightforward BFS

- We use *k* threads to visit neighbors
- We want to use as many threads as the number of neighbors

- Neighbors of U are not-contiguous in the Adjacency list array
- We want a contiguous array of neighbors.

Beyond the straightforward BFS

- We use *k* threads to visit neighbors
- We want to use as many threads as the number of neighbors

- Neighbors of U 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 overview

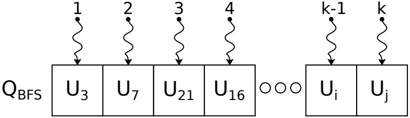
What we will do is:

1. Compute the total number of neighbors, say m
2. Start m threads, read the Adjacency list and build a contiguous array of neighbors (We call this array **Next Level Frontier**)
3. With m threads prune the **Next Level Frontier**
4. Exchange vertices with other processes and update the parent array

Sort-Uniq BFS

Recipe #1: compute the total number of neighbors

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

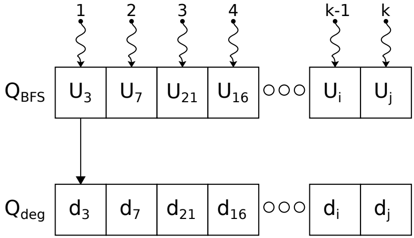


- Build Q_{deg}, substituting each vertex with its degree
- Perform a **prefix-sum** operation on Q_{deg} to build the **New Offset** array

Sort-Uniq BFS

Recipe #1: compute the total number of neighbors

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

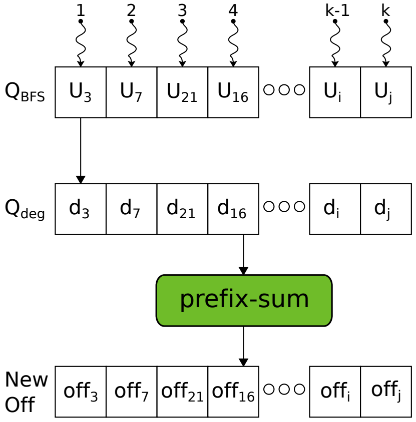


- Build Q_{deg}, substituting each vertex with its degree
- Perform a **prefix-sum** operation on Q_{deg} to build the **New Offset** array

Sort-Uniq BFS

Recipe #1: compute the total number of neighbors

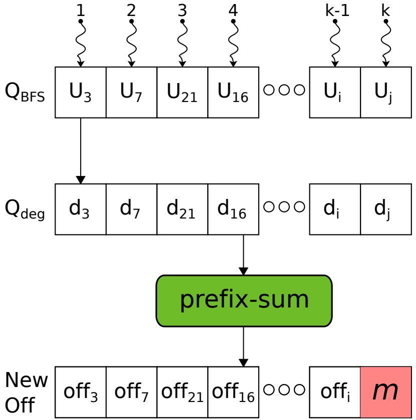
- Start *k* threads, each element of Q_{BFS} is assigned to one thread
- Build Q_{deg}, substituting each vertex with its degree
- Perform a **prefix-sum** operation on Q_{deg} to build the **New Offset** array



Sort-Uniq BFS

Recipe #1: compute the total number of neighbors

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

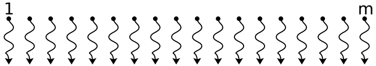


The last element of **New Offset** is: $m = \sum_{i \in Q_{BFS}} d_i$

Sort-Uniq BFS

Recipe #2: build a contiguos array of neighbors

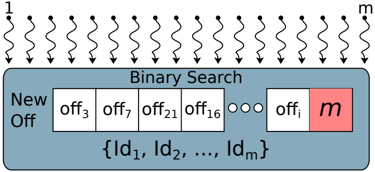
- Start *m* threads
- Each thread performs a **binary search** on **New Offset** and finds its index
- Each thread reads from the Adj list the element corresponding to the index
- and write it in the **Next Level Frontier**.



Sort-Uniq BFS

Recipe #2: build a contiguos array of neighbors

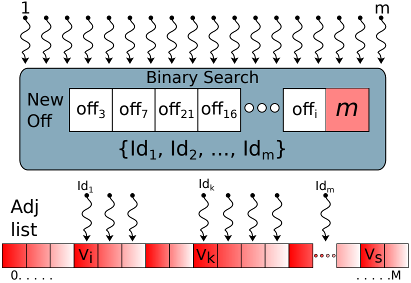
- Start *m* threads
- Each thread performs a **binary search** on **New Offset** and finds its index
- Each thread reads from the Adj list the element corresponding to the index
- and write it in the **Next Level Frontier**.



Sort-Uniq BFS

Recipe #2: build a contiguos array of neighbors

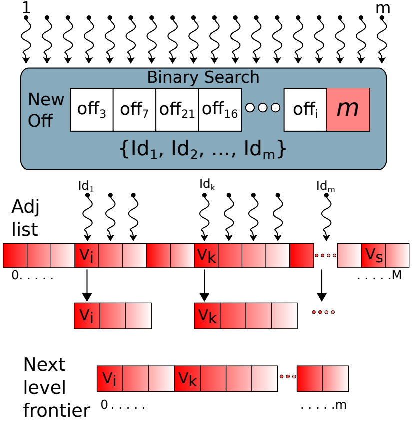
- Start *m* threads
- Each thread performs a **binary search** on **New Offset** and finds its index
- Each thread reads from the Adj list the element corresponding to the index
- and write it in the **Next Level Frontier**.



Sort-Uniq BFS

Recipe #2: build a contiguos array of neighbors

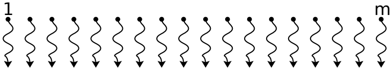
- Start *m* threads
- Each thread performs a **binary search** on **New Offset** and finds its index
- Each thread reads from the Adj list the element corresponding to the index
- and write it in the **Next Level Frontier**.



Sort-Uniq BFS

Recipe #3: prune the Next Level Frontier

- Start *m* threads

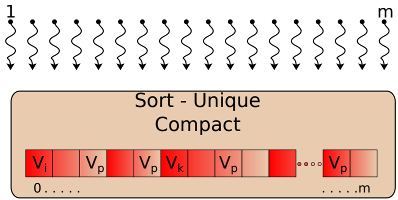


- Perform a **sort-uniq** operation on the **Next Level Frontier** (by using thrust library)
- and compact it to *n* unique elements

Sort-Uniq BFS

Recipe #3: prune the Next Level Frontier

- Start *m* threads

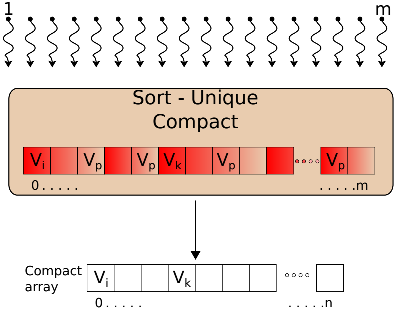


- Perform a **sort-uniq** operation on the **Next Level Frontier** (by using thrust library)
- and compact it to *n* unique elements

Sort-Uniq BFS

Recipe #3: prune the Next Level Frontier

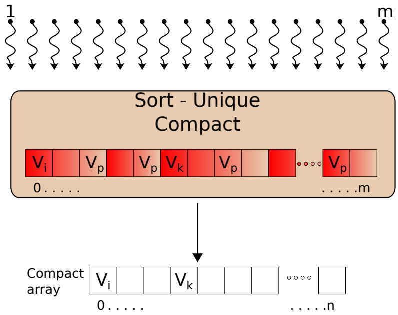
- Start *m* threads
- Perform a **sort-uniq** operation on the **Next Level Frontier** (by using thrust library)
- and compact it to *n* unique elements



Sort-Uniq BFS

Recipe #3: prune the Next Level Frontier

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

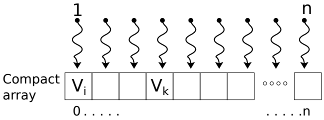


- Unique ratio $\frac{m}{n} \sim 20$

Sort-Uniq BFS: communication and enqueue

Recipe #4: Exchange vertices and update the parent array

- Start *n* threads

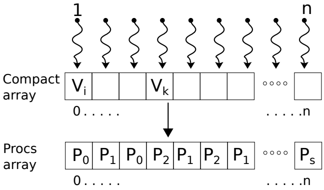


- Substitute vertices with tasks
- Sort by tasks (by using thrust library)
- Exchange non-local edges
- Update the array of predecessors and Enqueue

Sort-Uniq BFS: communication and enqueue

Recipe #4: Exchange vertices and update the parent array

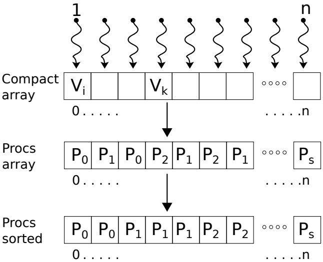
- Start *n* threads
- Substitute vertices with tasks
- Sort by tasks (by using thrust library)
- Exchange non-local edges
- Update the array of predecessors and Enqueue



Sort-Uniq BFS: communication and enqueue

Recipe #4: Exchange vertices and update the parent array

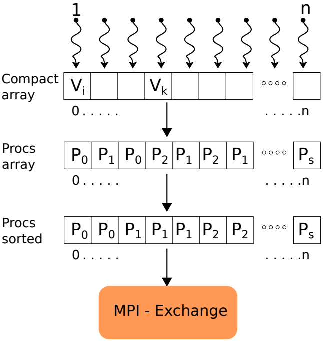
- Start *n* threads
- Substitute vertices with tasks
- Sort by tasks (by using thrust library)
- Exchange non-local edges
- Update the array of predecessors and Enqueue



Sort-Uniq BFS: communication and enqueue

Recipe #4: Exchange vertices and update the parent array

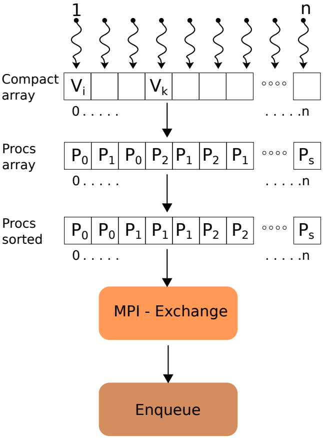
- Start *n* threads
- Substitute vertices with tasks
- Sort by tasks (by using thrust library)
- Exchange non-local edges
- Update the array of predecessors and Enqueue



Sort-Uniq BFS: communication and enqueue

Recipe #4: Exchange vertices and update the parent array

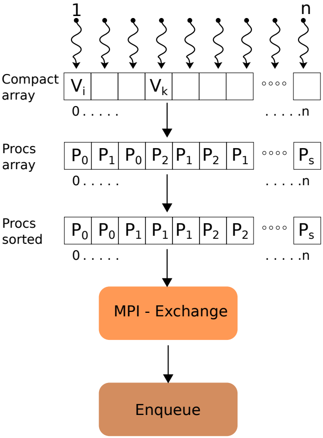
- Start *n* threads
- Substitute vertices with tasks
- Sort by tasks (by using thrust library)
- Exchange non-local edges
- Update the array of predecessors and Enqueue



Sort-Uniq BFS: communication and enqueue

Recipe #4: Exchange vertices and update the parent array

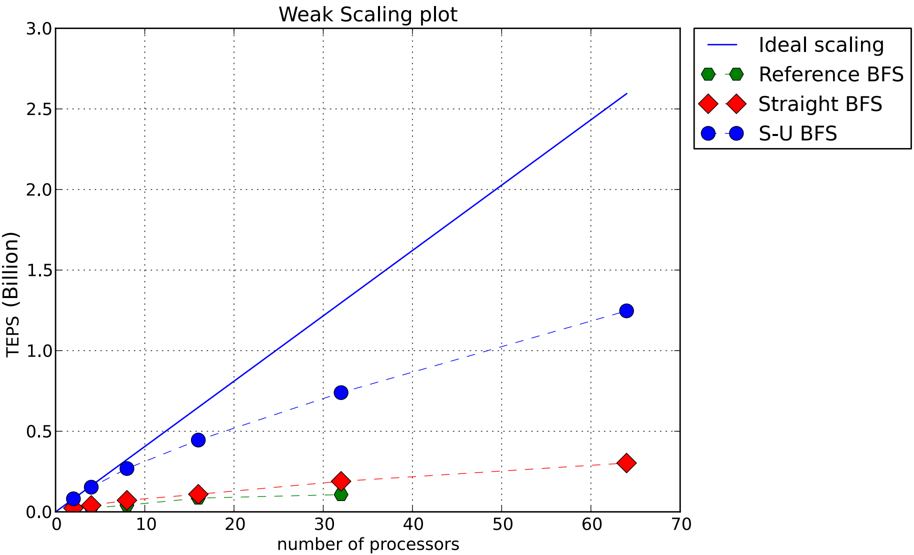
- Start *n* threads
- Substitute vertices with tasks
- Sort by tasks (by using thrust library)
- Exchange non-local edges
- Update the array of predecessors and Enqueue



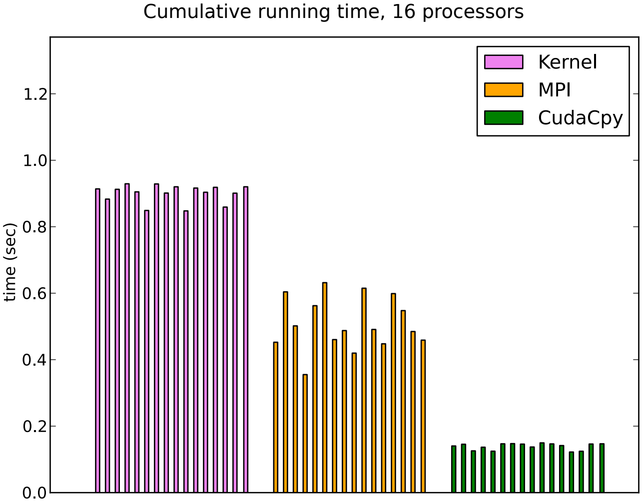
- If $Q_{BFS} == 0$ quit.

Outline

Sort-Uniq BFS: Results

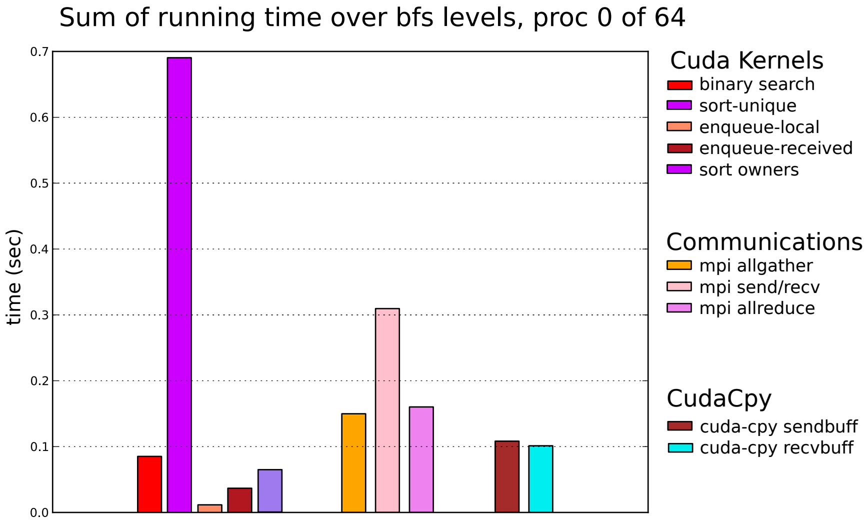


K2: balancing

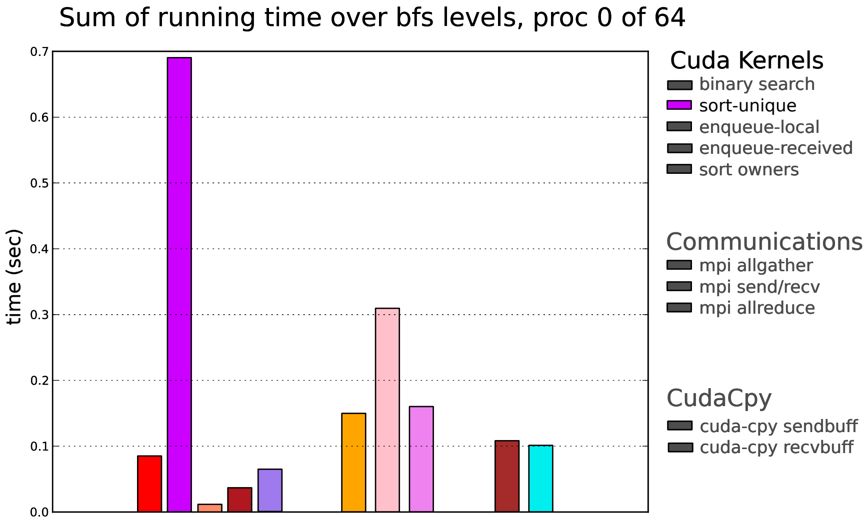


Computations and communications among processes are well balanced

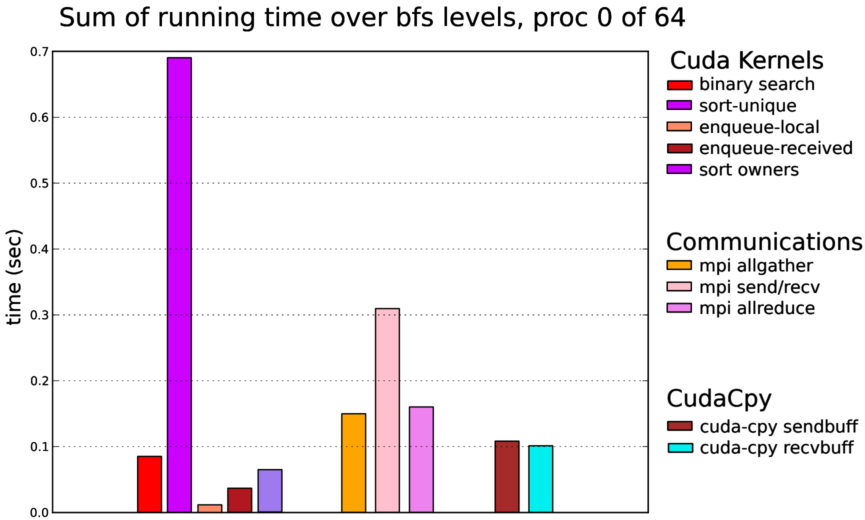
K2: cuda kernels times



K2: cuda kernels times



K2: cuda kernels times



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
4	Jugene (65,536 nodes)	Forschungszentrum Jülich	37	92,876,900,000	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

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
4	Jugene (65,536 nodes)	Forschungszentrum Jülich	37	92,876,900,000	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

Conclusions and Outlook

- To visit large graphs we need a distributed algorithm
- We are slower on a single GPU
- We relay on sorting to achieve better scaling
- If we can speed-up the sorting then we will speed up the BFS

D. Chakrabarti, D. Chakrabarti, Y. Zhan, and C. Faloutsos.
R-mat: A recursive model for graph mining.
IN SDM, 2004.

A. G. Duane Merrill, Michael Garland.
High performance and scalable gpu graph traversal.
Technical report, Nvidia, 2011.

P. Harish and P. J. Narayanan.
Accelerating large graph algorithms on the gpu using cuda, 2007.

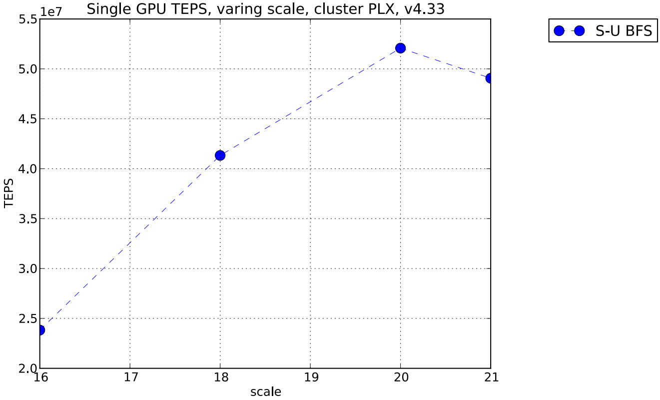
J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani.
Kronecker graphs: An approach to modeling networks.
J. Mach. Learn. Res., 11:985–1042, March 2010.

R. Murphy and G. Chukkapalli.
Graph 500 and data-intensive hpc.
Technical report, Oracle.

P. of the 16th ACM symposium on Principles and practice of parallel programming, editors.
Accelerating CUDA Graph Algorithms at Maximum Warp, 2011.

[5] [1, 4] [3, 6, 2]

Single GPU with the multi-GPUs code



Random Graph

Unique ratio example

2 0.28	2 0.47	2 0.18
3 0.16	3 0.15	3 0.21
4 0.77	4 0.50	4 0.97
5 1.00	5 1.00	5 1.00
6 1.00	6 1.00	6 1.00
	7 1.00	

Table: Unique ratio, proc 0 of 64, 3 run of BFS