Efficient Parallel Graph Exploration on Multi-Core CPU and GPU

Pervasive Parallelism Laboratory
Stanford University

Sungpack Hong, Tayo Oguntebi,
and Kunle Olukotun

Graph and its Applications

- Graph
  - Fundamental data structure
  - G = (N,E): *Arbitrary* relationship (E) between data entities (N)
- Wide range of Applications
  - Scheduling task graphs
  - PDE (Partial Differential Equation) solver on mesh
  - Artificial Intelligence – Bayesian network
  - Bioinformatics – molecular interaction graph
  - Social network analysis
  - Web graphs
  - Graph database – schema-less data management

Requires large graph analysis

Performance Issues

- Single-core machines showed limited performance for large graph analysis problems
  - A lot of random memory accesses

    \+ Data does not fit in cache

    ➔ Performance is bound to memory latency
  - Conventional hardware units (e.g. floating point, branch predictors, out-of-order) do not help much

➔ Use parallelism to accelerate graph analysis
  - Plenty of data-parallelism in large graph instances
    - Latency bound ➔ Bandwidth bound
  - Exploit recent proliferation of parallel computers:

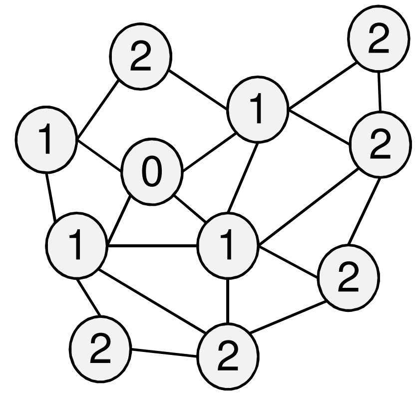
    Multi-core CPU and *GPU*

Graph Exploration

- Breadth first search (BFS)
  - A systematic way to traverse the graph
  - A building block for many other algorithms
    - s-t connectivity, betweeness centrality, connected component, community detection, max-flow …
  - Can be parallelized (c.f. depth first search)
    - More about this in the next slide
  - Many previous researches on implementation
    - For various architectures: Cluster, Cell, Cray, Multi-core/SMP, GPU, …
  - Preferred as parallel benchmark
    - See graph500.org

Parallel BFS Algorithm

- Start from a root, and visit all the connected nodes in a graph
- Nodes closer to the root are visited first
- Nodes of the same hop-distance (level) from the root can be visited in parallel



Algorithm 1 Level Synchronous Parallel BFS

procedure BFS(r:Node)
  $V = C = \emptyset$; $N = \{r\}$ ▷ Visited, Current, and Next set
  r.lev $= level = 0$
  repeat
    $C = N$
    for Node $c \in C$ **do** ▷ in parallel
      for Node $n \in$ Nbr(c) **do** ▷ in parallel
        if $n \notin V$ **then**
          $N = N \cup \{n\}$; $V = V \cup \{n\}$
          n.lev $= level + 1$
    $level$++
  until $N = \emptyset$

Three Node-sets

Nodes of the current level

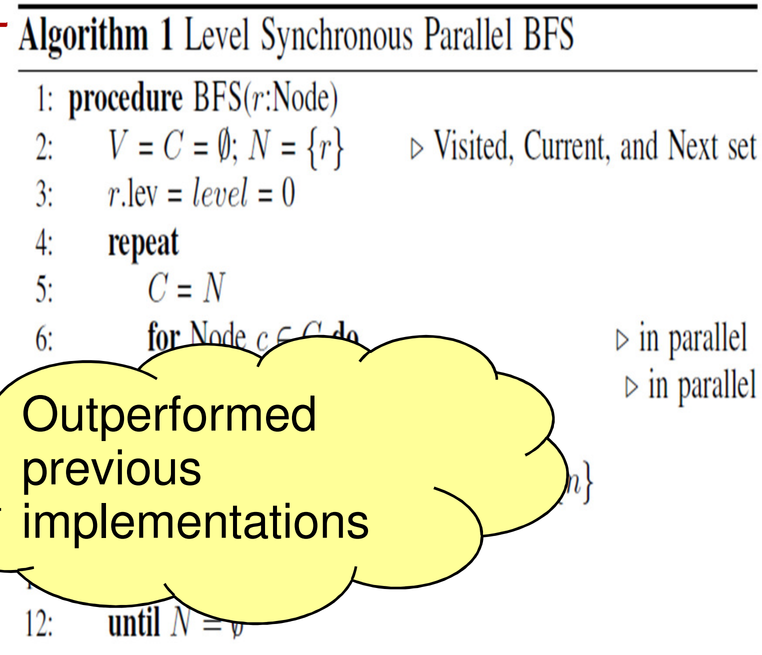
Neighbors of current level nodes

Add non-visited neighbors to Next and Visited set

Synchronization at the end of each level

Implementation for Multi-Core CPU

- Level Synchronous Parallel BFS
  - Requires synchronization at everyl evel
  - Degree of parallelism limited by (# nodes) in each level
- State-of-Art Implementation
  - [Agarwal et. al. SC 2010]
  - V ➔ bitmap
    - Maximize cache hit ratio
    - Atomic update required: 'test and test-and-set'
  - C, N ➔ queue
    - Local Queue + Global Queue
    - Complex queue implementation based on ticket-lock and fast forwarding
      - Not so much details revealed in their paper
    - Avoid unnecessary cache-to-cache traffic


Algorithm 1 Level Synchronous Parallel BFS

```
procedure BFS(r:Node)
    V = C = ∅; N = {r}          ▷ Visited, Current, and Next set
    r.lev = level = 0
    repeat
        C = N
        for Node c ∈ C do                    ▷ in parallel
                                              ▷ in parallel
        ...n}
    until N = ∅
```

Outperformed previous implementations

Can we do better?

- Issues
  - Requires complex queue implementation
  - Can we do better even without it?
- Our two implementations
- Queue-Based Implementation
  - Approximate Agarwal et. al.'s approach
    - Bitmap
    - Test and Test-and-Set
    - Local Q + Global Q
    - *Standard Queue*
- Another implementation
  - Exploit properties of the graphs
  - Exploit properties of the machines

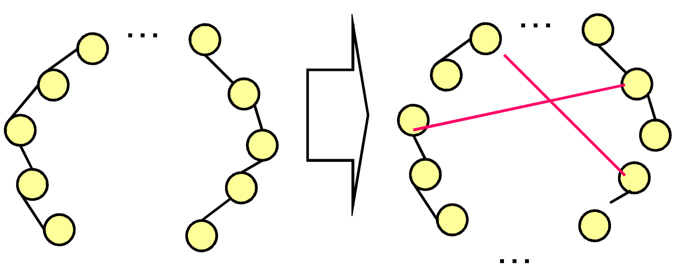
Observation on Graphs

- Small-World Property [Watts and Strogatz, Nature 1998]
  - Any randomly-shaped graphs has a small diameter *("Six-degrees of separation")*
  - A fundamental property
    : web graphs, social graphs, molecular graphs, …

[Corollary] There must be at least one level that has O(N) nodes.

Regular Graph:
Diameter → O(N)

Adding Ramdom re-wiring:
Diameter → O(c)



Total execution time is governed by these *critical* levels

Level	Num. Nodes
0	1
1	4
2	749
3	109,239
4	7,103,690
5	9,088,766
6	130,298
7	172

(e.g.) Number of nodes at each BFS level (16 million node graph)

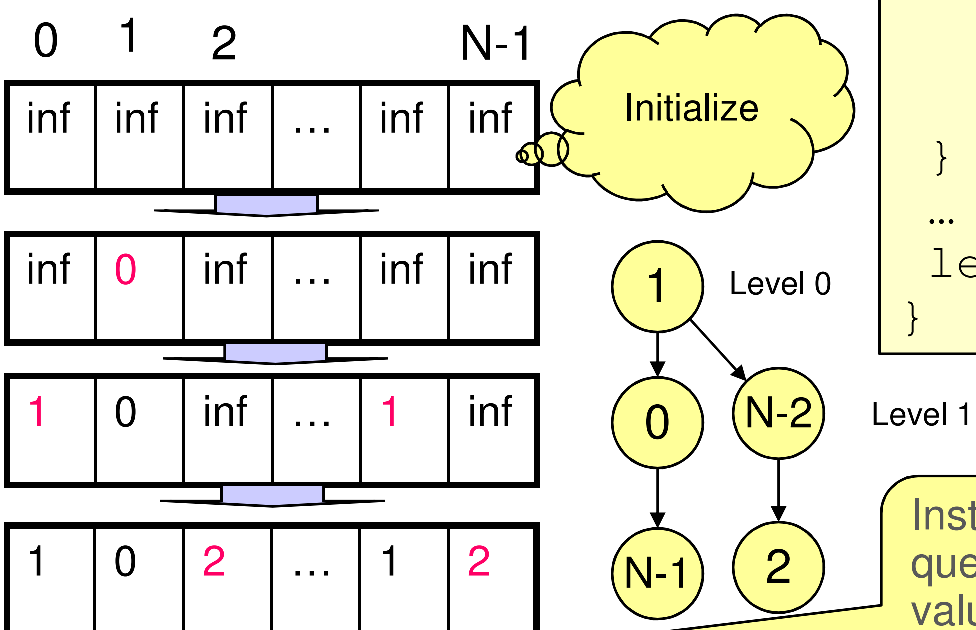
Read-based implementation

- Another implementation of ours
- V: Bitmap
- C, N: Level-Array
  - A single O(N)-sized array that keeps the level of each node

Read the entire array!

```
...
while (!finished) {
  foreach (c: G.Nodes) {
    if (level[c] != curr_lev)
      continue;
      ...
  }
  ...
  lev++;
}
```

Iterate nodes in Current set



Instead of keeping queues, update the value in the level array.

Adding nodes to Next set

What's the benefit of that?

(1) The array is read sequentially

(1)-b Overall access pattern become more sequential as well

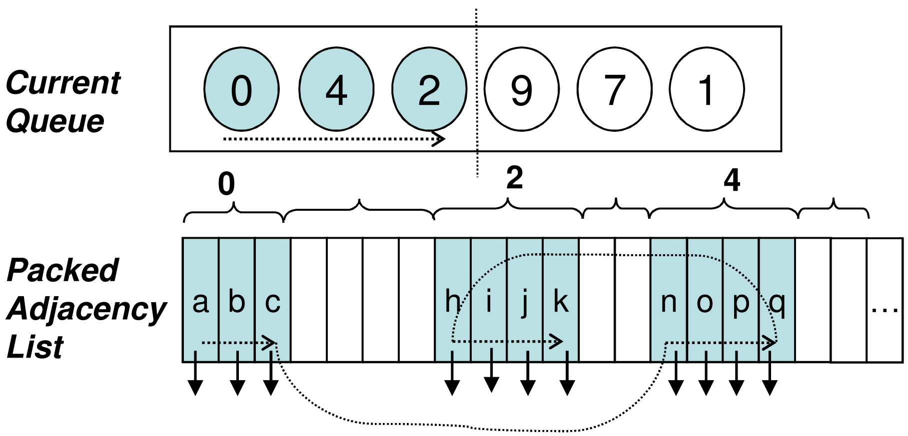
(2) There are only a few level;

In critical levels, you have to visit O(N) nodes anyway.

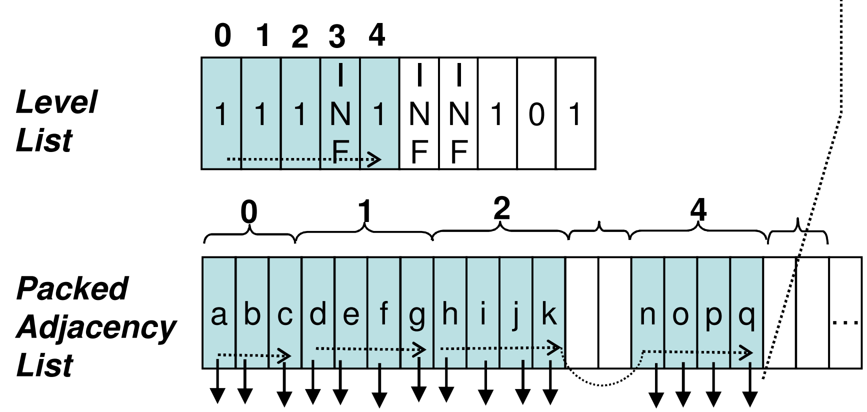
10x Diff

Machine	Seq. Read	Random Read
Nehalem CPU	8.6 GB/s	0.98 GB/s
Core CPU	3.0 GB/s	0.25 GB/s
Fermi GPU	76.8 GB/s	2.71 GB/s
Tesla GPU	72.5 GB/s	3.15 GB/s

But cannot eliminate all the natural random accesses.



(a) Data-Access Pattern of Queue-Based Method



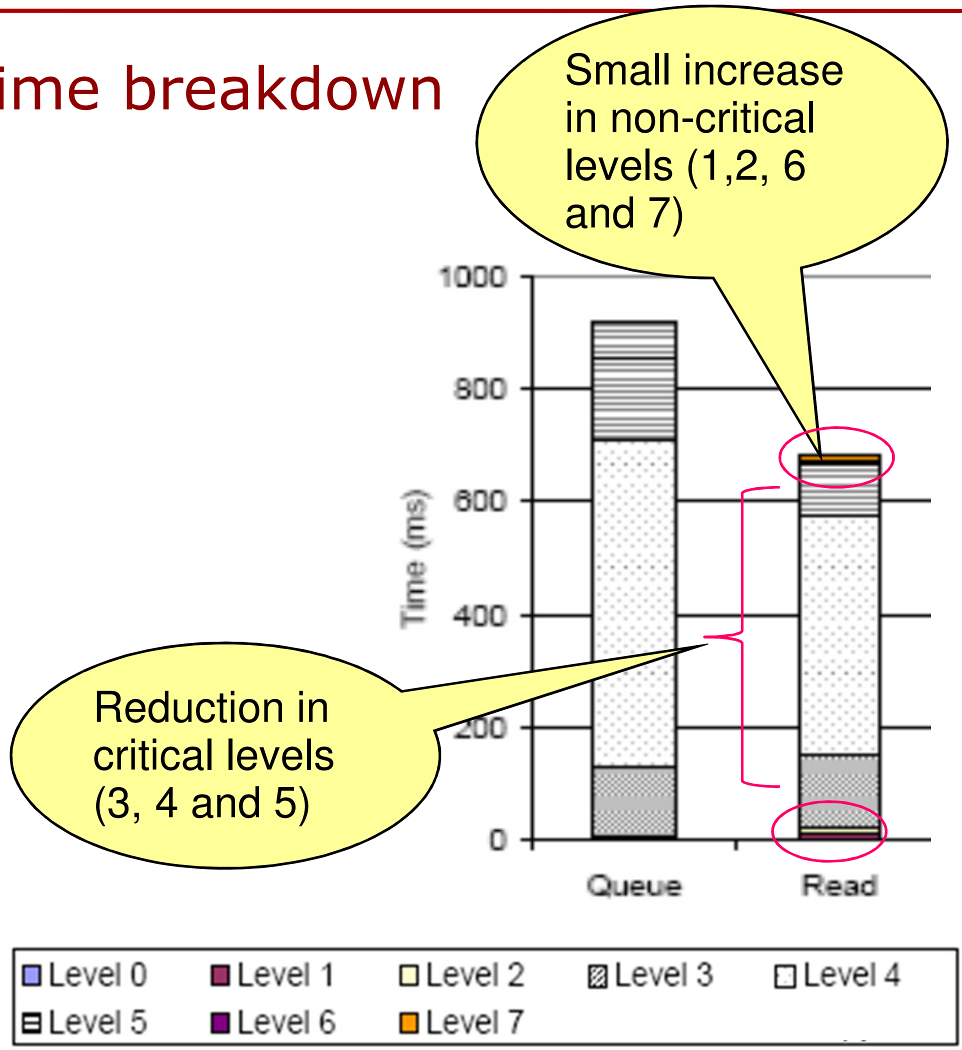
(b) Data-Access Pattern of Read-Based Method

Queue-Based vs. Read-Based

- Level-wise execution time breakdown

Level	Num. Nodes
0	1
1	4
2	749
3	109,239
4	7,103,690
5	9,088,766
6	130,298
7	172

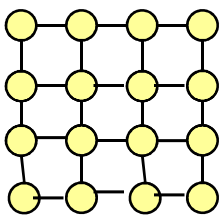
(e.g.) Number of nodes at each BFS level (16 million node graph)

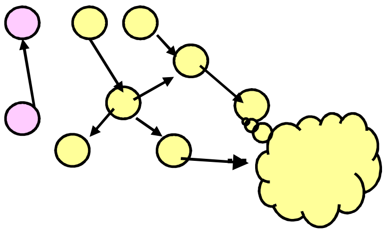


What about big-world graphs?

- Worst-case inputs for Read-based method:

1. High-diameter graphs
   - Recent graph applications (e.g. social network) deal with small-world graphs more frequently
   - Still, there are high-diameter graphs: e.g. mesh





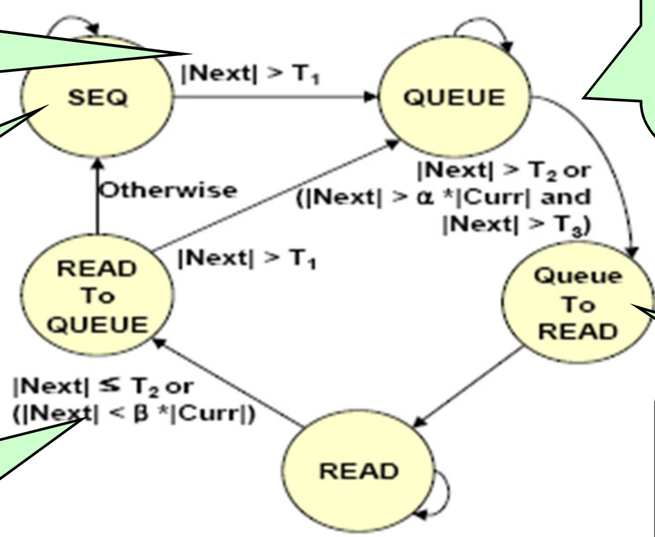
2. Small search instance
   - When the graph is not (strongly) connected
   - Your traversal finishes after visiting only small portion of the graph

Preventing worst case execution

- Our solution: hybrid method
  - Choose appropriate method (Read or Queue), adaptively at each level
  - Based on the size of Next set and its growth rate.
- Finite State Machine

Go Parallel (with Queues) when there are enough # nodes.

If Next is large enough (e.g. p% of num nodes) or exponential growing, migrate to Read method



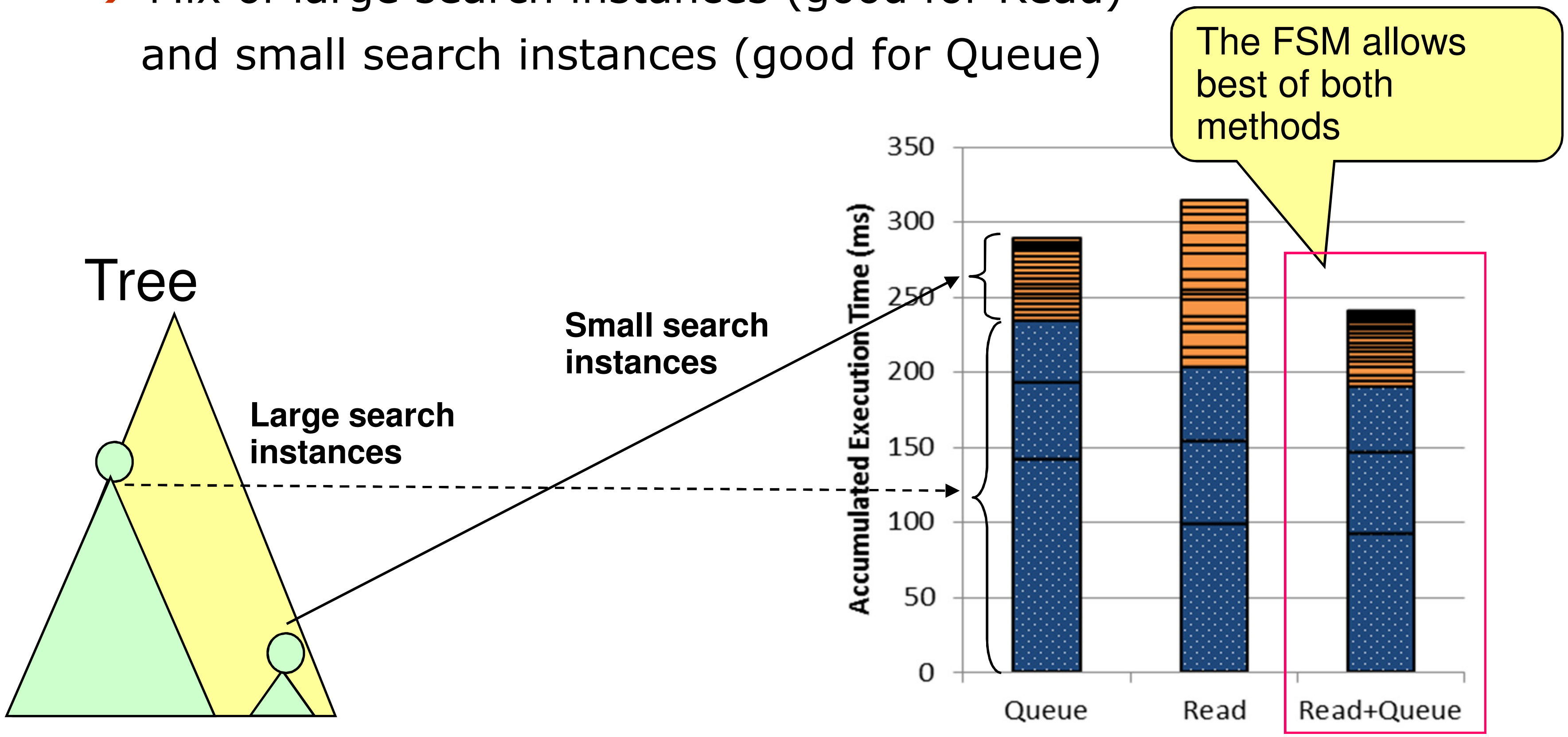
Process the root node, sequentially.

Return to Queue method when Next is shrinking

Transient state:
Read from Queue,
Write to Array

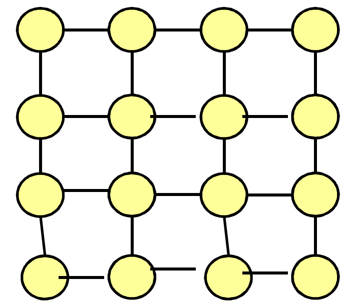
Result: worst-case avoidance

- BFS on tree
  - Y-axis: time (high is bad)
  - Mix of large search instances (good for Read) and small search instances (good for Queue)

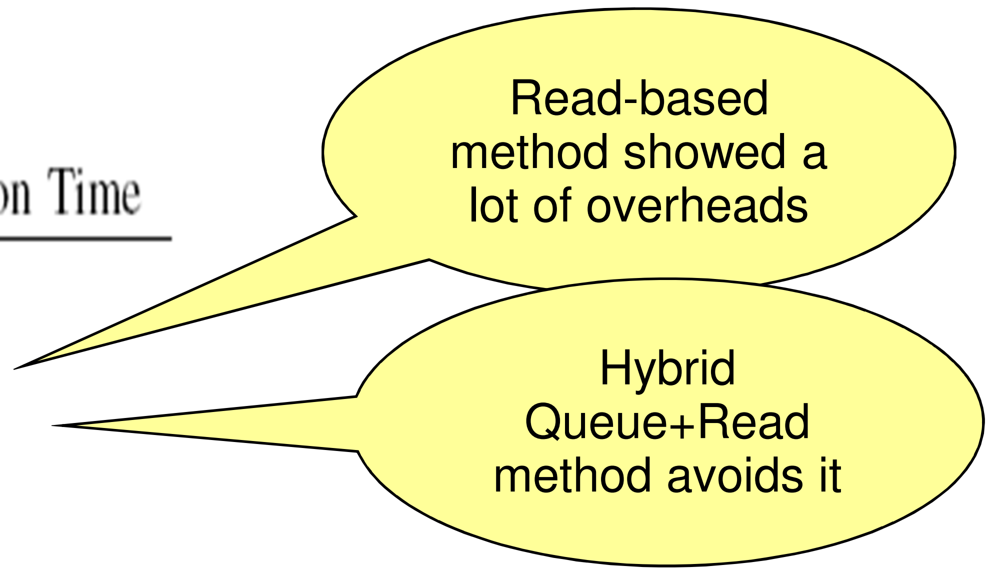


Result: worst-case avoidance

- 2-D Mesh
  - 4000x4000
  - Diameter is O(sqrt(N))
    - (# nodes) at each level increases not exponentially, but linearly



Method	Normalized Execution Time
Queue	1.00
Read	12.63
Queue+Read	1.01

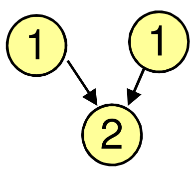


Graph Exploration on GPU

- GPU Benefits
  - Large memory bandwidth (GDDR, # channels)
  - Massively parallel hardware
    - HW multi-threading + SIMD(/SIMT)
  - HW Traits similar to Cray-XMT
    - But much cheaper
- GPU Issues
  - Limited capacity (~ a few GB)
- Our approach:
  - Use GPU, only if the graph fits
  - Use multi-core CPU, otherwise
  - But how much performance does this give?

Graph Exploration on GPU

- BFS on GPU
  - [Harish and Narayanan, HiPC 2007], [Hong et al, PPoPP2011]
  - Similar to Queue-based implementation
  - Visited, Next, Current ➔ Level Array
    - If level[node] is INF, then node is not visited
    - Hard to do bitwise atomic operation efficiently on GPU
  - A node can be written multiple times by different parents ➔ Okay, because the written level value is always same

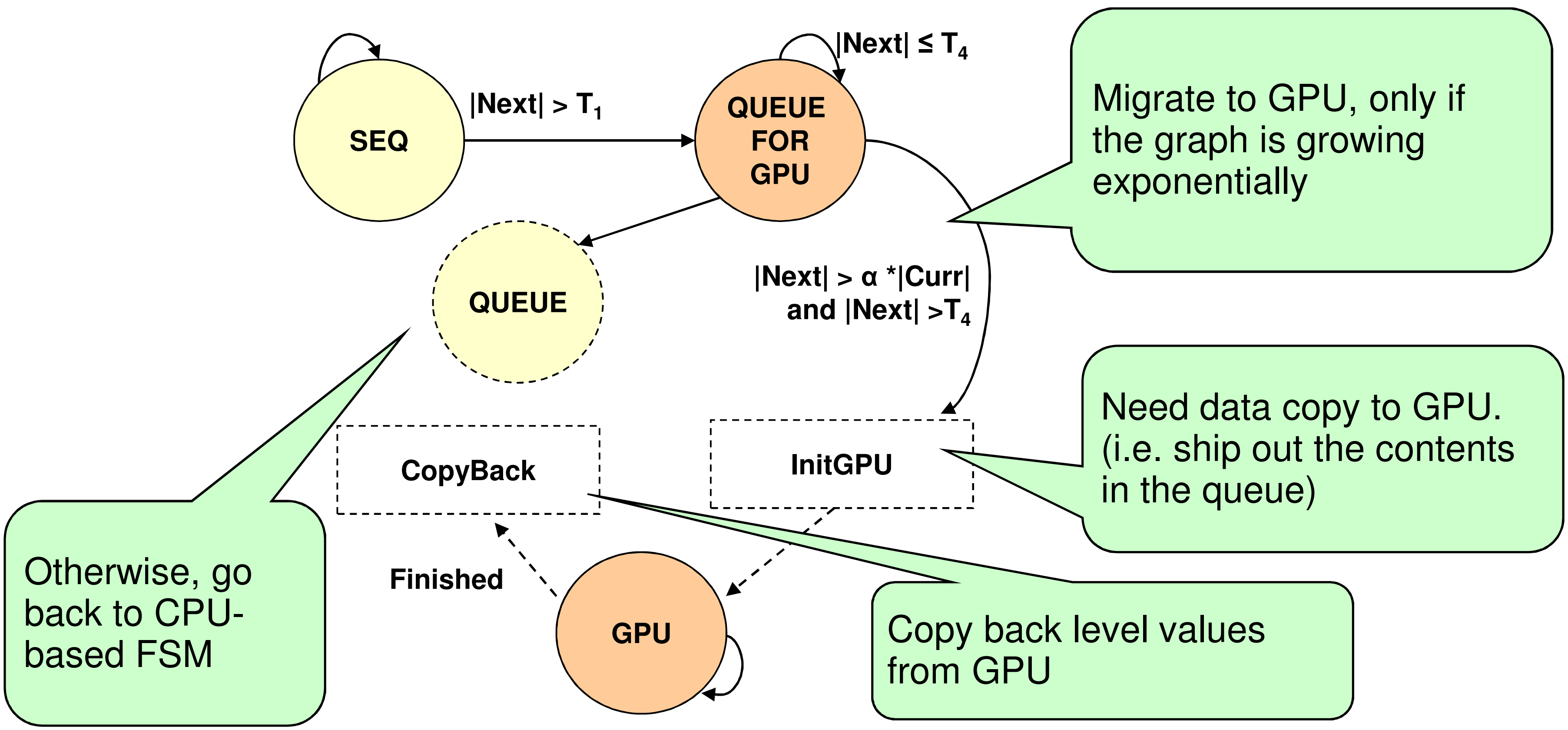


  - ... But it has the same issue as Queue-based method

    ➔ Bad for small or long-diameter graphs

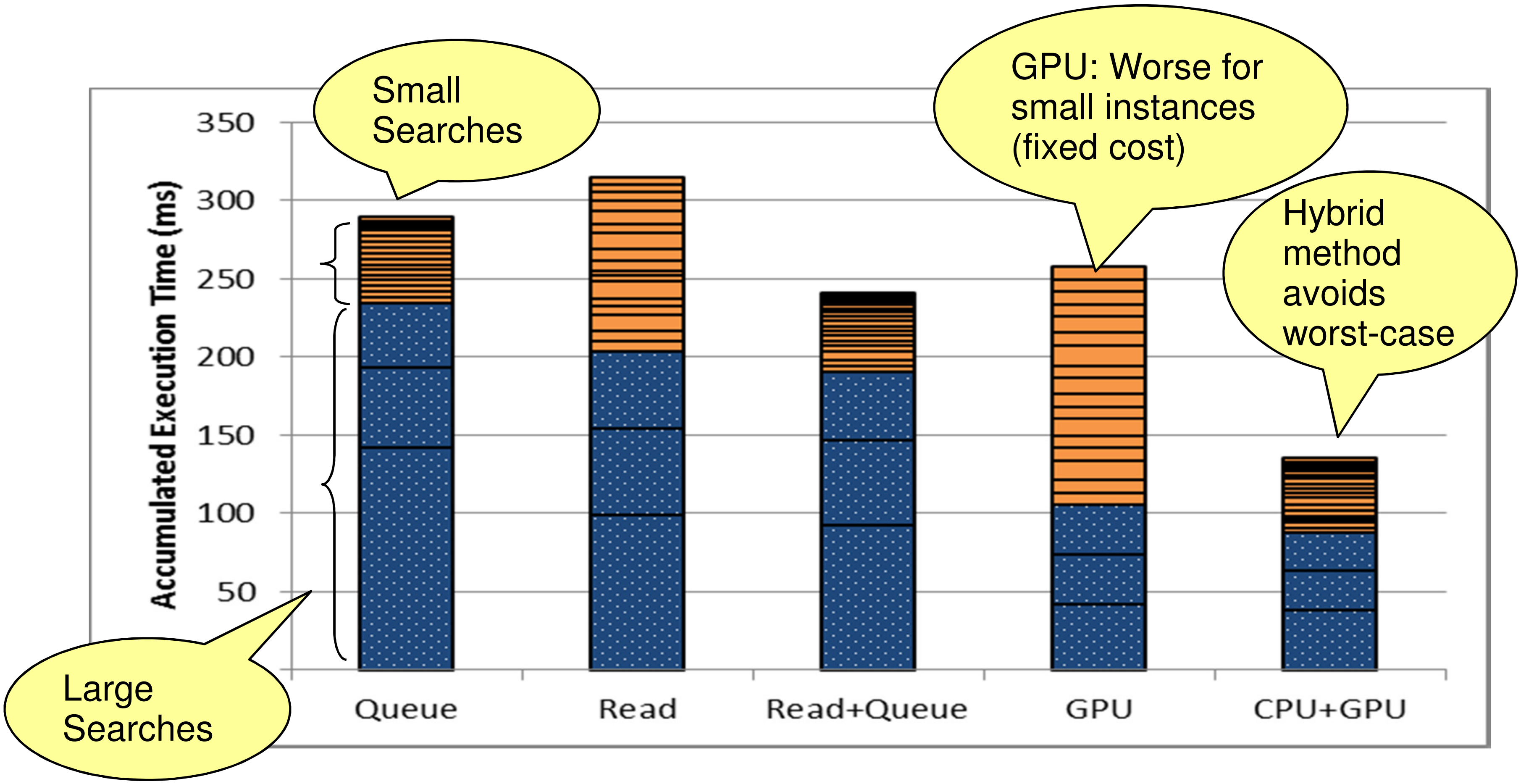
Hybrid CPU+GPU

- An extension to the previous FSM



GPU: Worst-case avoidance

- BFS on tree with GPU

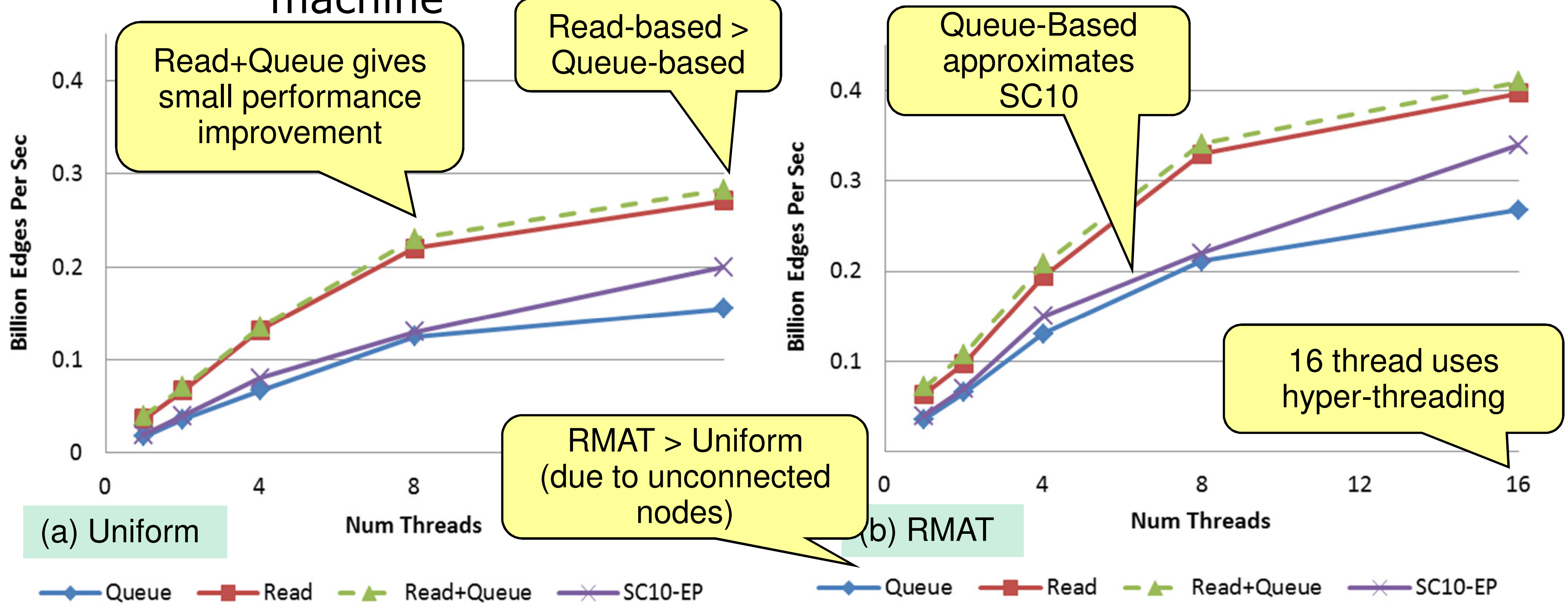


Experiments on Small-world Graphs

- Multi-Core CPU
  - Intel Nehalem (X5550) 2.67GHz
  - 2 Socket x 4 Core x 2 HT
  - LLC: 8MB x 2
  - Main Memory: 24GB
- GPU
  - Nvidia Fermi (C2050) 1.15GHz
  - 14 SM x (2 warps) x 32 SIMT
  - LLC: 2MB
  - Main Memory: 3GB
- Measurement
  - Start from multiple root nodes
  - Measure average execution time from multiple executions
- Graphs
  - Two kinds of widely accepted synthetic graphs
  - Random (Erdos-Renyi)
    - Simple uniform random
  - RMAT
    - Skewed degree distribution (good)
    - Many (~50%) unconnected nodes (bad)
  - 32mil nodes, 256 mil edges

Performance Results

- Multi-Core CPU Result
  - y-axis: processing rate (Higher is better)
  - SC10-EP: numbers from [Agarwal et. al SC10]
    - Measured for same sized graph on a faster (2.9Ghz) machine

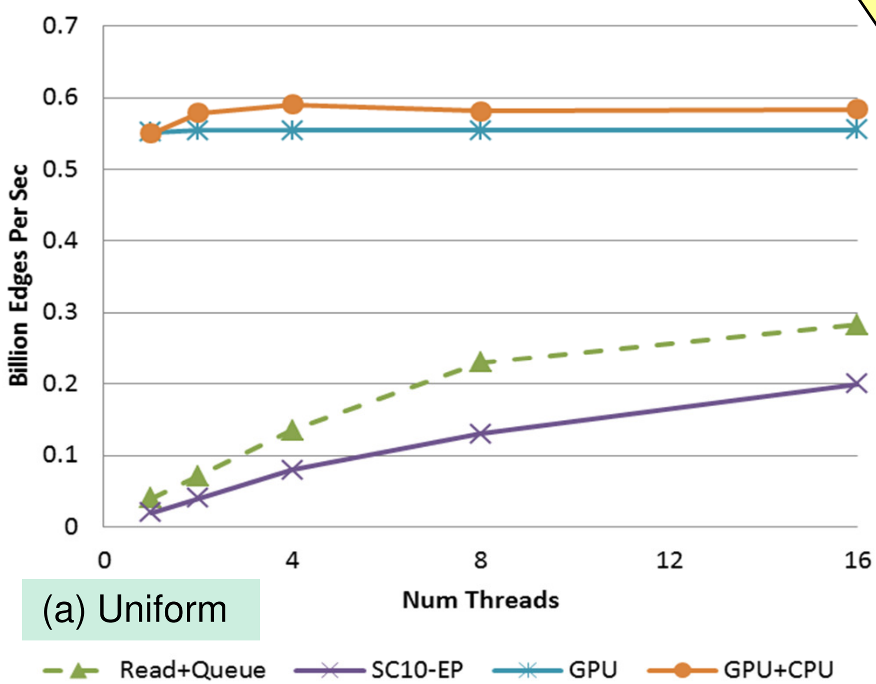


Performance Results

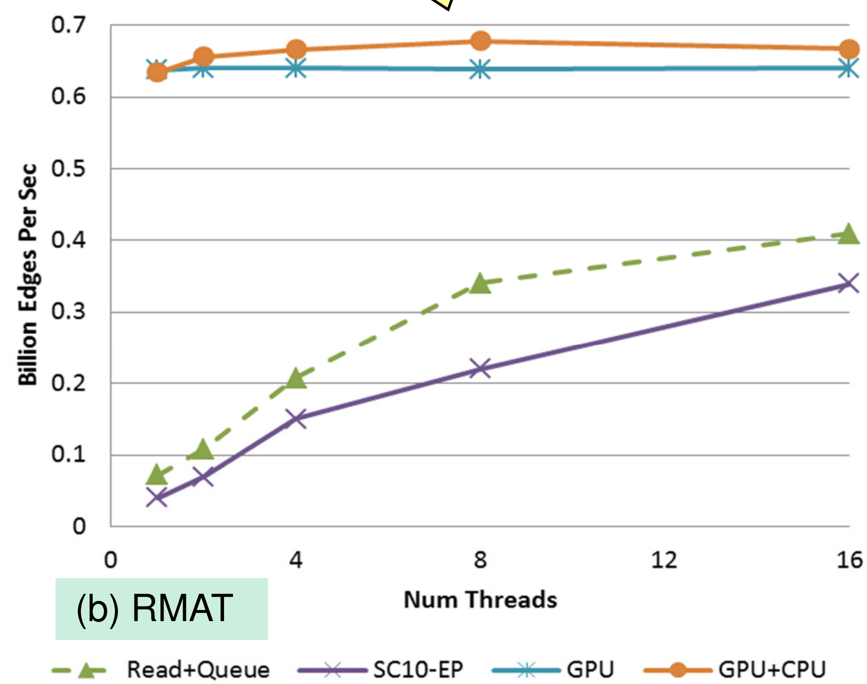
- GPU Result
  - Same graph inputs

GPU:1.5x ~ 2x (compared to best CPU 16 threads)

CPU + GPU can give small performance improvement



(a) Uniform

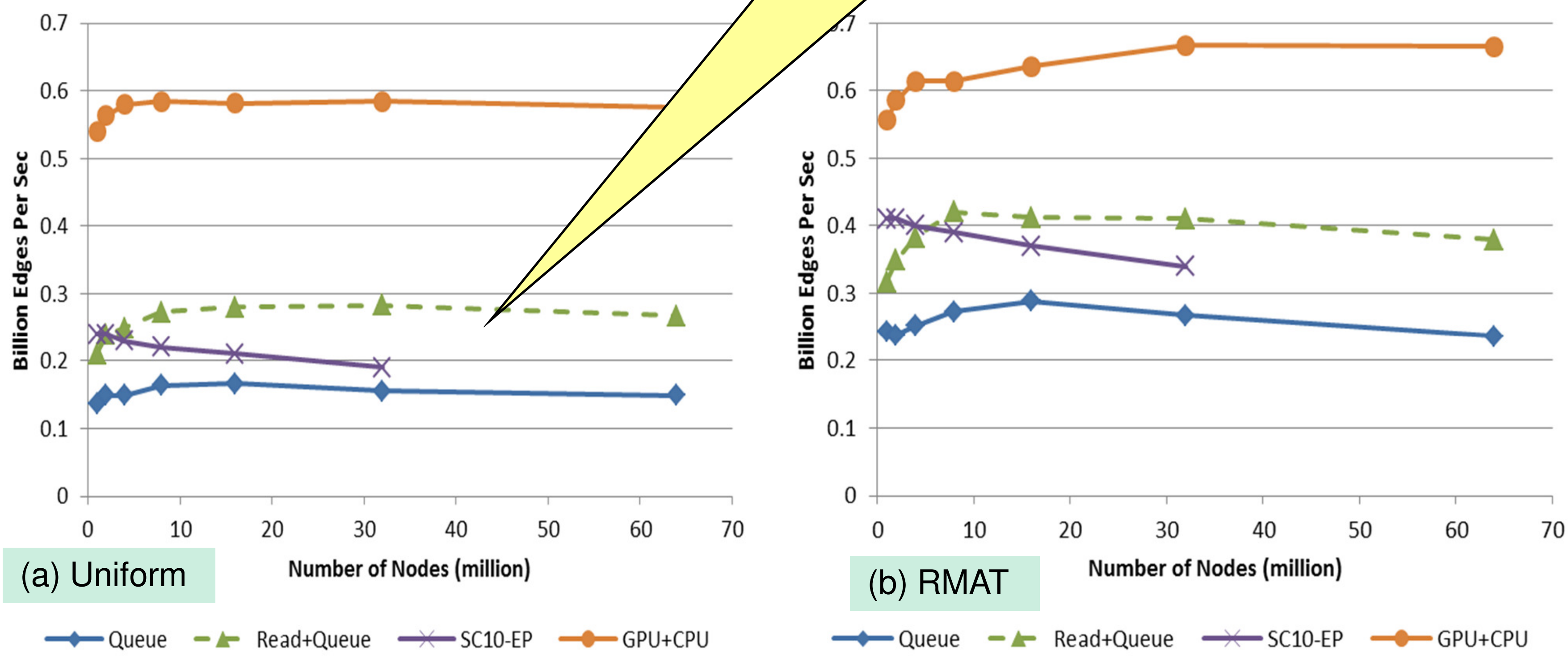


(b) RMAT

Changing Graph Size

- Varying number of nodes
  - 1mil ~ 64 mil
  - # Edges = (# Nodes) x 8
  - # Threads = 16

Performance difference widens as graph size grows (cache-cache miss doesn't matter much)

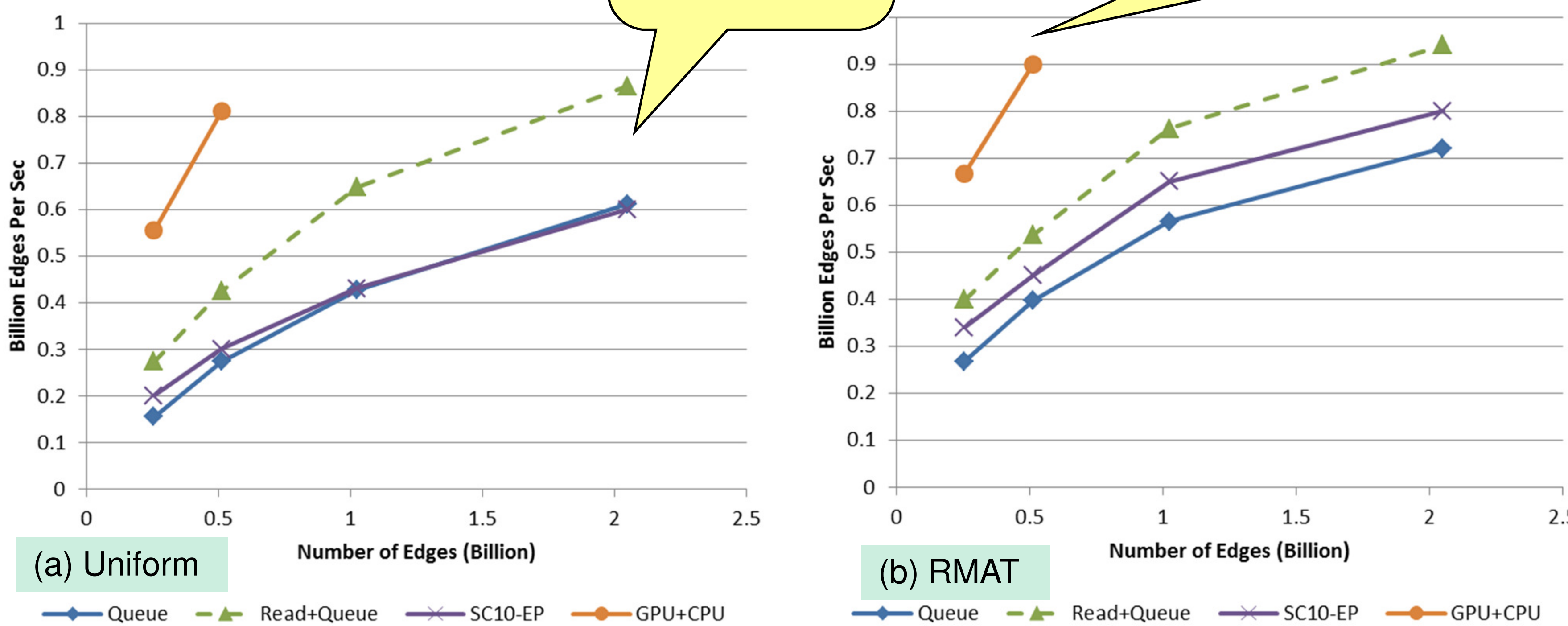


Changing Graph Size

- Varying number of edges
  - 256 mil ~ 2048 mil
  - # Nodes = 32 mil
  - # Threads = 16

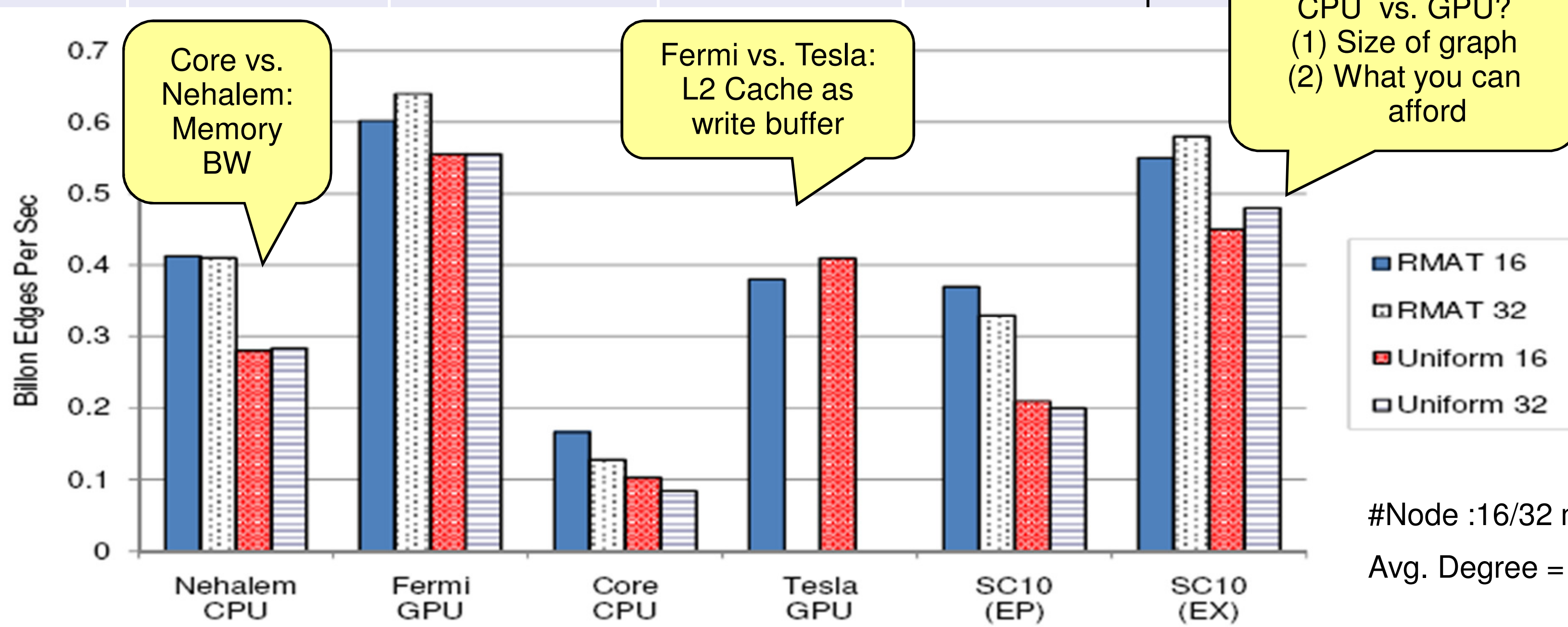
Performance gap widens as the graph size grows

GPU still performs better; but has hit size limit



Architectural Effects

					Nehalem	Nehalem
	Nehalem	**Fermi**	**Core**	**Tesla**	**SC10-EP**	**SC10-EX**
Freq.	2.67GHz	1.15GHz	2.33GHz	1.40GHz	2.93GHz	2.26GHz
(# Cores)	2 x 4 (x 2)	14 x 2	2 x 4	30	2 x 4 (x 2)	4 x 8 (x 2)
SIMD/SIMT	-	32	-	32	-	-
LLC (MB)	16 MB	2 MB	8 MB	-	16 MB	96 MB
Memory	24 GB	3 GB	32 GB	896 MB	48 GB	256 GB
Rnd Read	0.98 GB/s	2.71 GB/s	0.25 GB/s	3.15 GB/s	-	



#Node :16/32 mil

Avg. Degree = 8

Summary

- "Why" rather than "How"
- Exploited properties of graphs and machines
  - Small-world property
  - Bandwidth difference between sequential access and random access
- A simple state-machine to avoid worst-case execution
- Graph exploration on GPU
  - Limited capacity
  - Faster execution due to memory bandwidth

Thank you

- Questions?