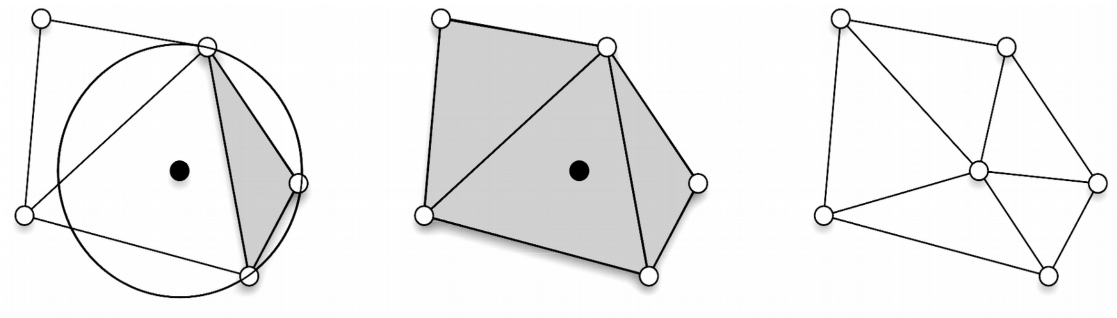
Parallel Graph Algorithms



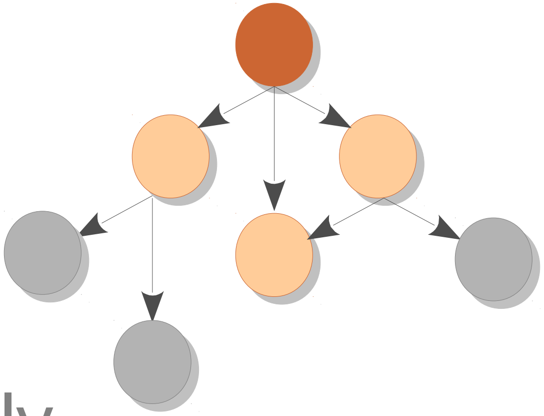
Rupesh Nasre.

Programming Languages, Architecture and Compilers Education Laboratory

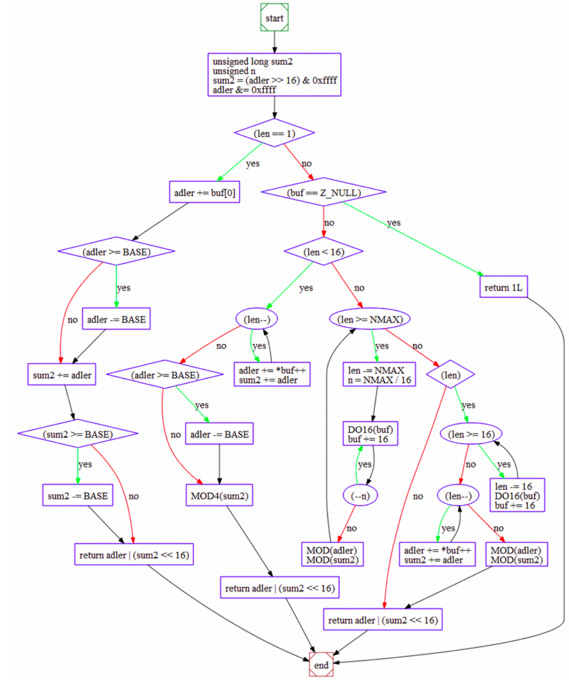
GPU Programming
January 2020

Graphs

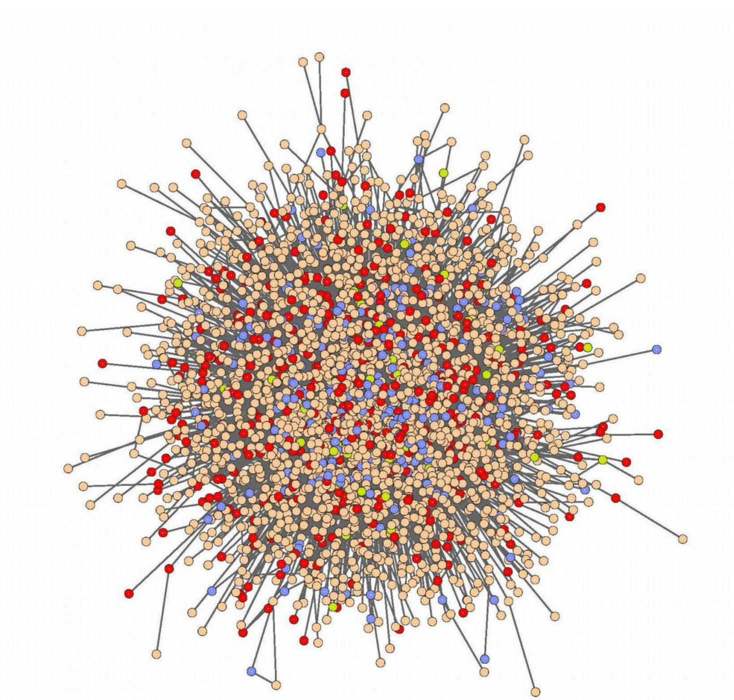
- Where do we encounter graphs?
  - Social networks, road connections, molecular interactions, planetary forces, …
  - snap, florida, dimacs, konect, ...



- Why treat them separately?
  - They provide structural information.
  - They can be processed more efficiently.
- What challenges do they pose?
  - Load imbalance, poor locality, …
  - Irregularity



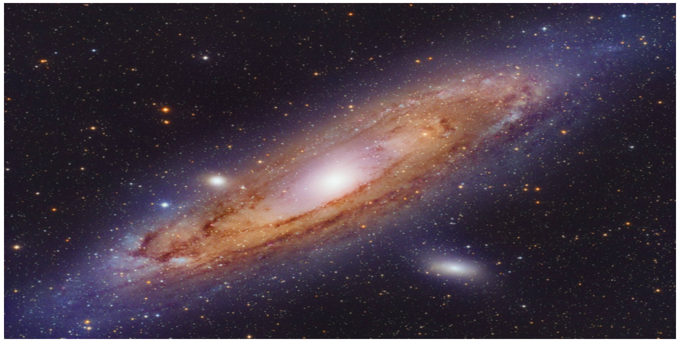
Graphs are Everywhere!

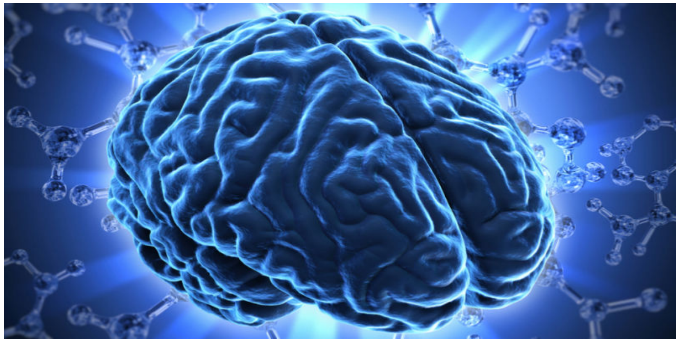


Source: Google images

Scalability

- **Facebook**
  - 2.2 billion active users
  - 1.3 billion is India's population
  - e.g. top people in the world
- **Milky Way**
  - over 100 billion stars
  - e.g. finding possibility of life
- **Human Brain**
  - 100 billion neurons
  - Artificial intelligence





Source: google images

```
Finding betweenness centrality on a million node graph (in a
sequential manner) takes several weeks!
```

Handling Large Graphs

Storage

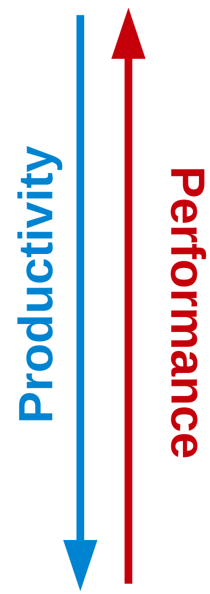
- Distributed setup
  - Graph is partitioned across a cluster.
- External memory algorithms
  - Graph partitions are processed sequentially.
- Algorithms on compressed data
  - Compression needs to maintain retrieval ability.
- Maintaining graph core
  - Removal of unnecessary subgraphs.

Time

- Parallelism
  - Multi-core, distributed, GPUs
- Approximations
  - Approximate computing

Parallelism Approaches

- Manual
- Libraries
  - Galois, Ligra, LonestarGPU, Gunrock, ...
- Domain-Specific Languages
  - Green-Marl, Elixir, Falcon, ...



Specifying Parallelism

- Do not specify.
  - Sequential input, completely automated, currently very challenging in general
- Implicit parallelism
  - aggregates, aggregate functions, primitive-based processing, ...
- Explicit parallelism
  - pthreads, MPI, CUDA, ...

Identifying Dependence

```
for (ii = 0; ii < 10; ++ii) {
    a[2 * ii] = ... a[2 * ii + 1] ...
}
```

Is there a flow dependence between different iterations?

Dependence equations

$0 \le ii_w < ii_r < 10$

$2 * ii_w = 2 * ii_r + 1$

which can be written as

$$\left.\begin{array}{l} 0 \le ii_w \\ ii_w \le ii_r - 1 \\ ii_r \le 9 \\ 2 * ii_w \le 2 * ii_r + 1 \\ 2 * ii_r + 1 \le 2 * ii_w \end{array}\right\} \quad \begin{pmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \\ 2 & -2 \\ -2 & 2 \end{pmatrix} \begin{pmatrix} ii_w \\ ii_r \end{pmatrix} \le \begin{pmatrix} 0 \\ -1 \\ 9 \\ 1 \\ -1 \end{pmatrix}$$

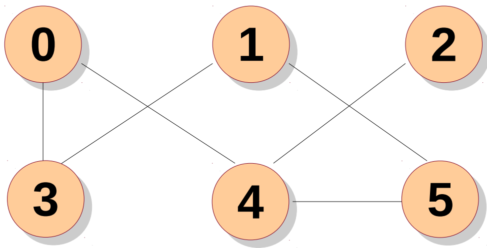
Dependence exists if the system has a solution.

Parallel Architectures

- **Multicore CPUs**
  - Intel, ARM, …
  - pthreads, OpenMP, ...
- **Distributed systems**
  - GraphLab, GraphX, …
  - MPI
- **Manycore GPUs**
  - NVIDIA, AMD, …
  - CUDA, OpenCL, ...

Challenges in Graph Algorithms

- Synchronization
  - locks are prohibitively expensive on GPUs
  - atomic instructions quickly become expensive
- Memory latency
  - locality is difficult to exploit
  - low caching support
- Thread-divergence
  - work done per node varies with graph structure
- Uncoalesced memory accesses
  - warp-threads access arbitrary graph elements



Graph Representation

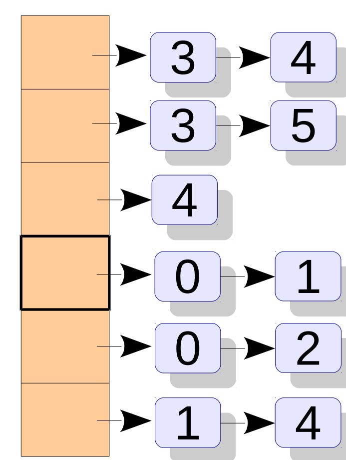
1. Adjacency matrix

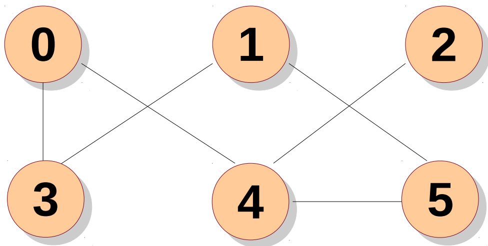
- |V| x |V| matrix
- Each entry [i, j] denotes if edge (i,j) is present in G
- Useful for **dense** graph
- Finding neighbors is O(|V|)

			1	1	
			1		1
				1	
1	1				
1		1			
	1			1	

2. Adjacency list

- |V| + |E| size
- Each vertex i has a list of its neighbors
- Useful for **sparse** graphs
- Finding neighbors is O(max. degree)





Graph Representation

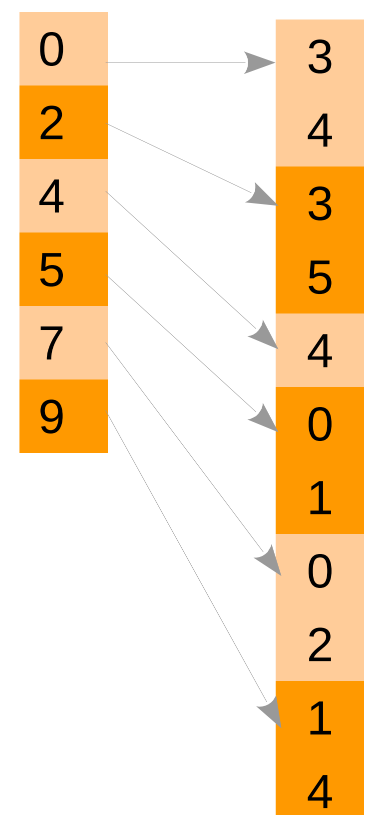
0	3
0	4
1	3
1	5
2	4
3	0
3	1
4	2
5	1
5	4

3. Edge list / Coordinate list (COO)

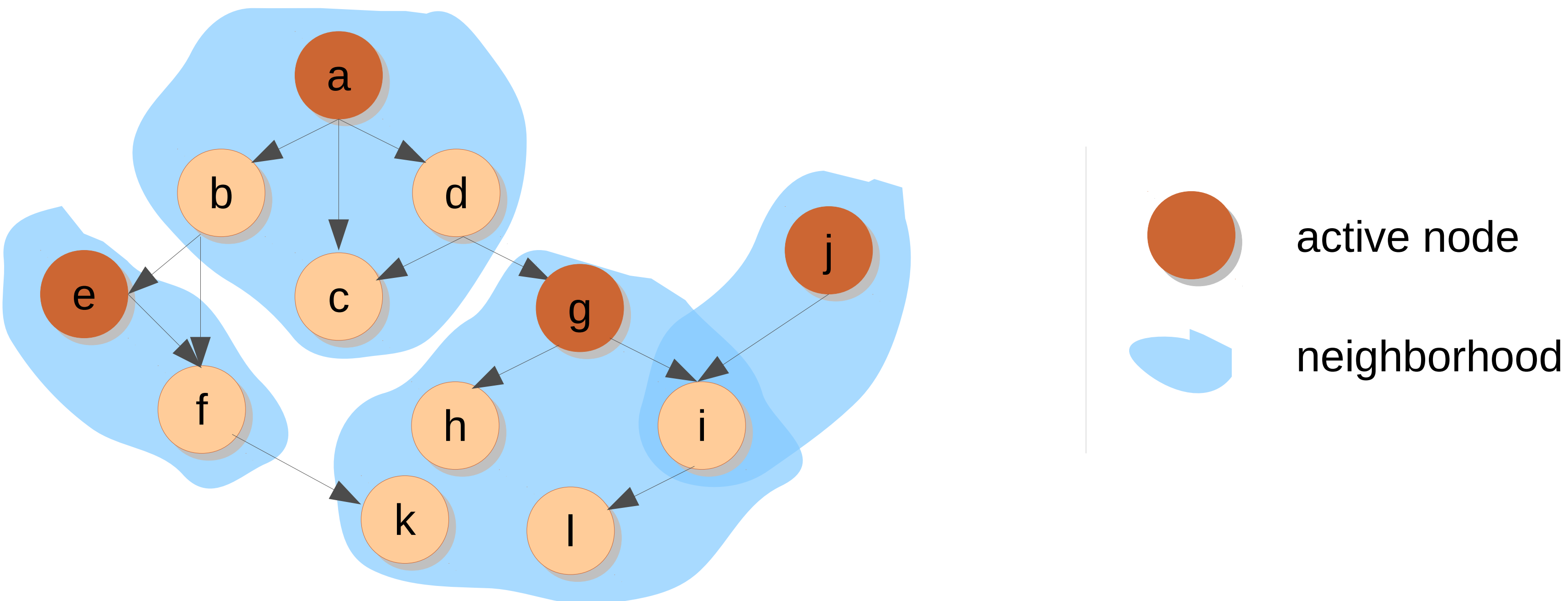
- $|E|$ pairs
- Useful for edge-based algorithms
- Typically sorted on vertex id

4. Compressed sparse row (CSR)

- Concatenated adjacency lists
- Useful for **sparse** graphs
- Useful for data transfer



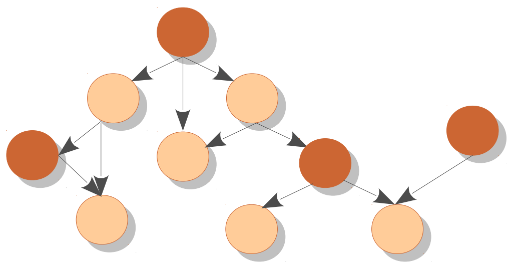
TAO Classification



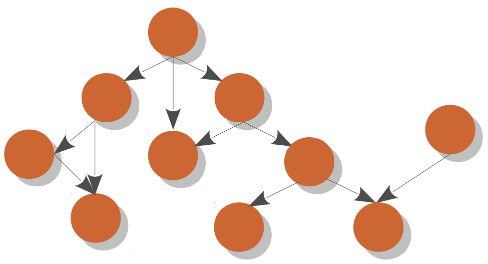
- **Operator formulation**: Computation as an iterated application of operator
- **Topology-driven processing**: operator is applied at all the nodes even if there is no work to do at some nodes (e.g., Bellman-Ford SSSP)
- **Data-driven processing**: operator is applied only at the nodes where there might be work to be done (e.g., SSSP with delta-stepping)

The TAO of Parallelism in Algorithms, Pingali *et al*, PLDI 2011

Data-driven vs. Topology-driven



data-driven



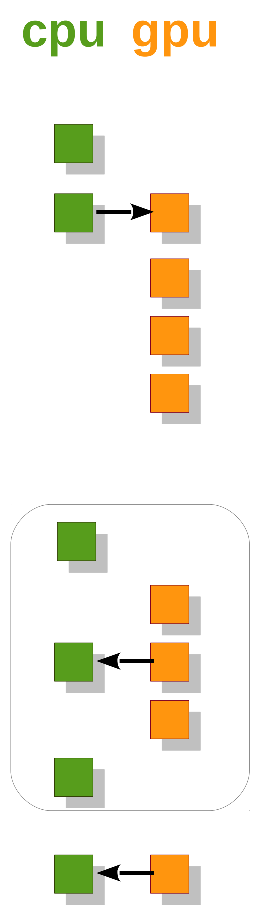
topology-driven

- work-efficient
- centralized worklist
- fine-grained synchronization using atomics
- complicates implementation

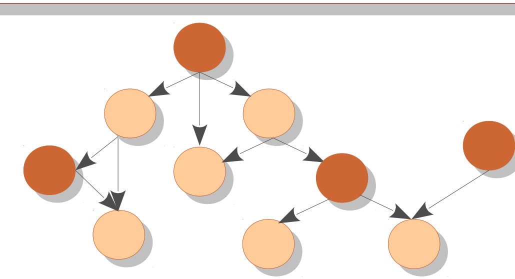
- performs extra work
- no worklists
- coarse-grained synchronization using barriers
- easier to implement

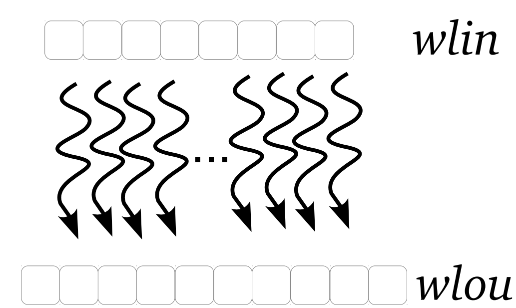
Data-driven: Base Version

```
main {
    read input
    transfer input
    initialize_kernel
    initialize_worklist(wlin)
    clear wlout

    while wlin not empty {
        operator(wlin, wlout, ...)
        transfer wlout size
        clear wlin
        swap(wlin, wlout)
    }
    transfer results
}
```

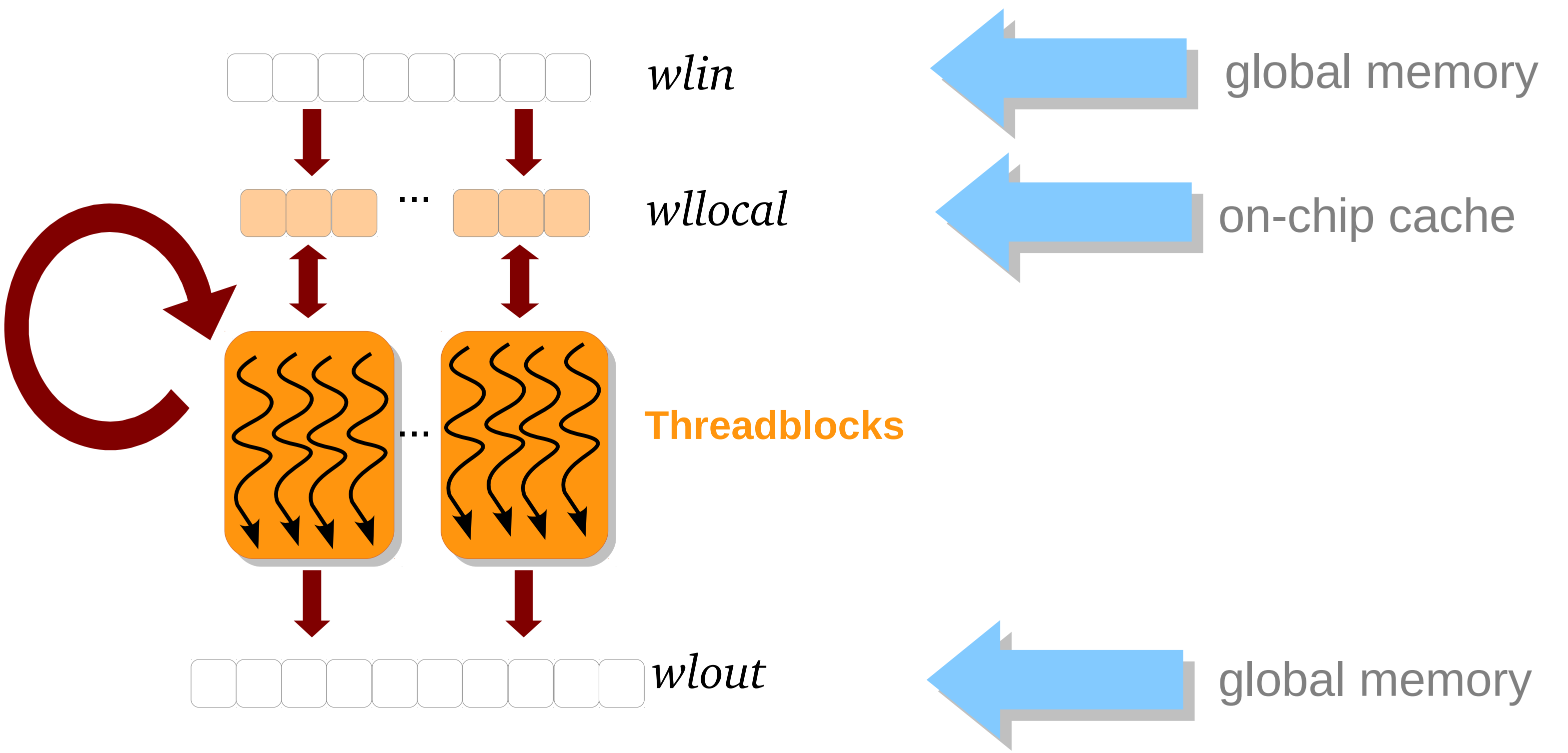



```
sssp_operator(wlin, wlout, ...) {
    src = wlin[...]
    dsrc = distance[src]
    forall edges (src, dst, wt) {
        ddst = distance[dst]
        altdist = dsrc + wt

        if altdist < ddst
            distance[dst] = altdist
            wlout.push(dst)
} }
```



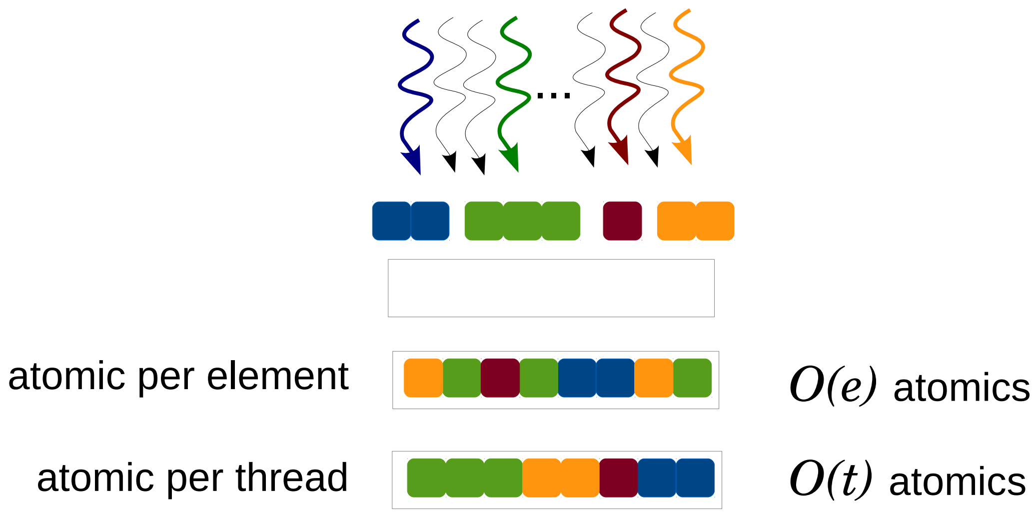


Data-driven: Hierarchical Worklist



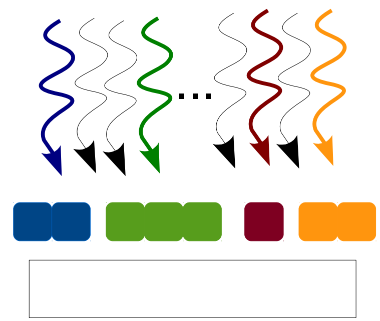
- Worklist exploits memory hierarchy
- Makes judicious use of limited on-chip cache

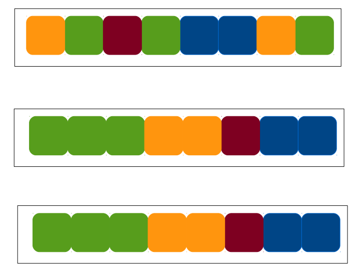
Data-driven: Work Chunking



- Reserves space for multiple work-items in a single atomic
- May reduce overall synchronization

Data-driven: Atomic-free Worklist Update

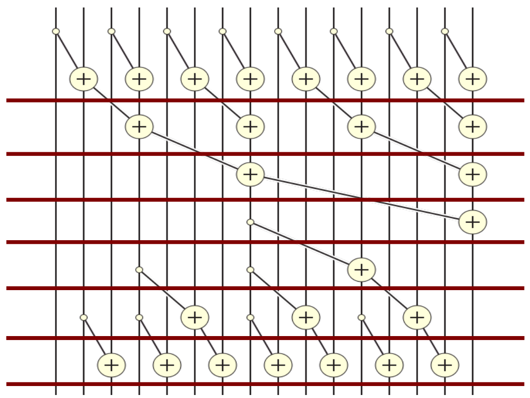




atomic per element — $O(e)$ atomics

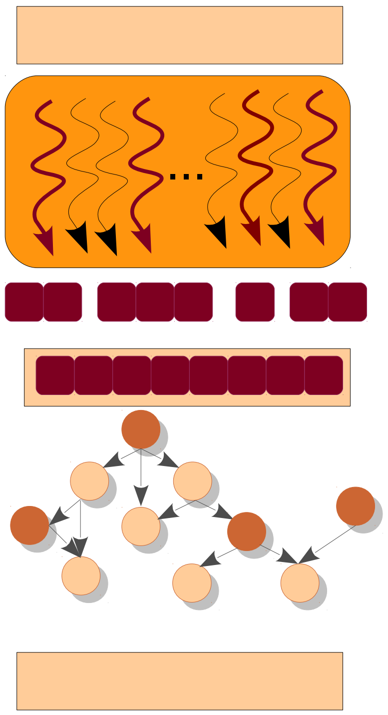
atomic per thread — $O(t)$ atomics

prefix-sum — $O(\log t)$ barriers



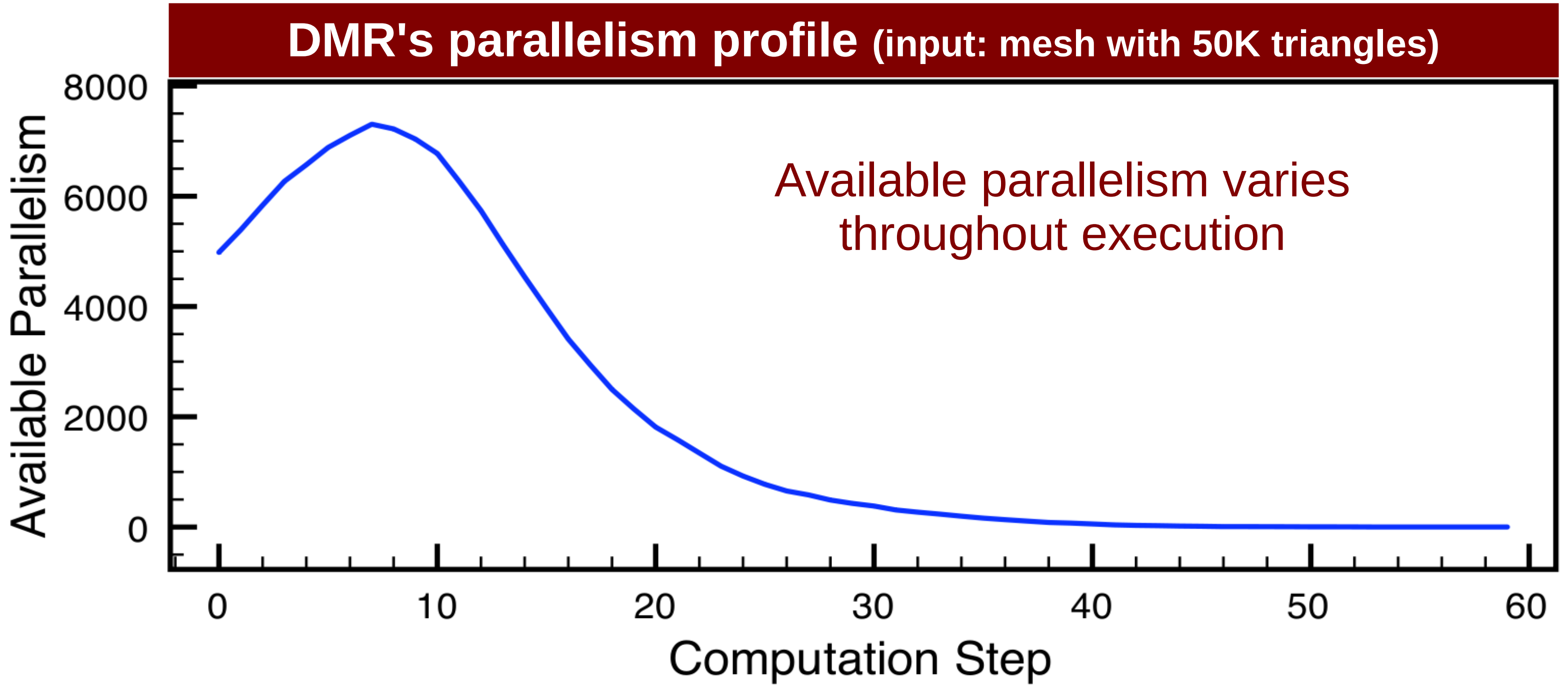
Data-driven: Work Donation

```
donate_kernel {
      shared donationbox[...];


      // determine if I should donate
      --barrier--

      // donate
      --barrier--

      // operator execution


      // empty donation box

}
```



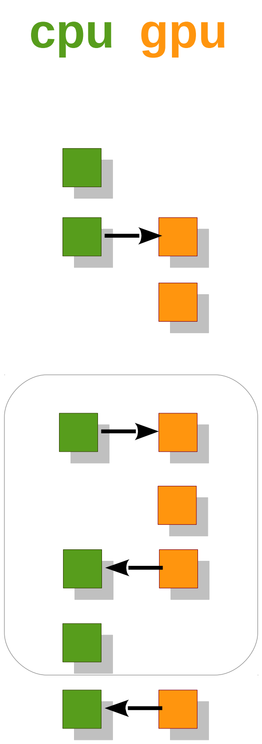
- Work-donation improves load balance

Data-driven: Variable Kernel Configuration

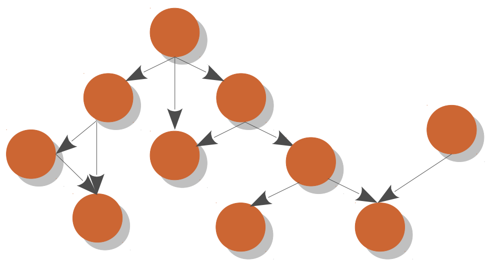


- Varying configuration improves work-efficiency
- It also reduces conflicts and may improve performance

Topology-driven: Base Version



main {
    read input
    transfer input
    initialize_kernel
    do {
        transfer **false** to *changed*
        operator(...)
        transfer *changed*
    } **while** *changed*
    transfer results
}



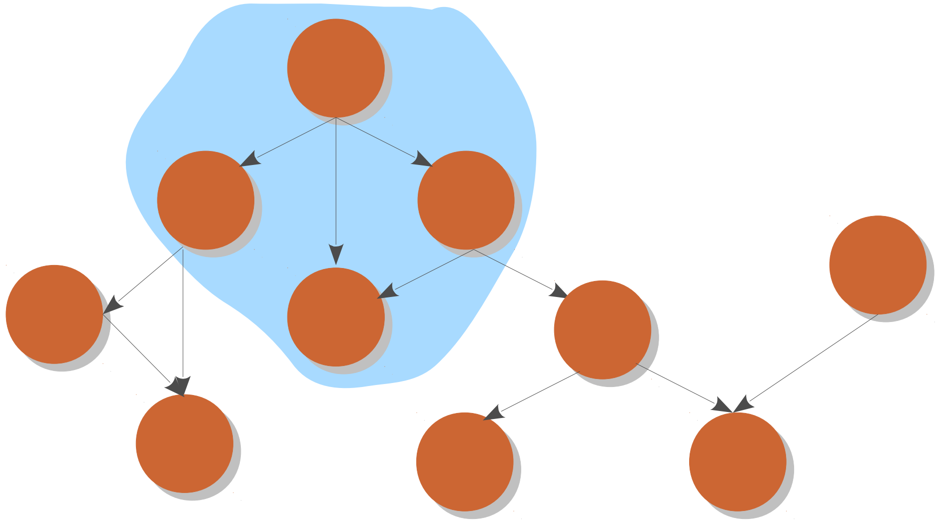
Topology-driven: Kernel Unrolling

Memory-bound kernel

sssp_operator(*src*) {
    $dsrc = distance[src]$

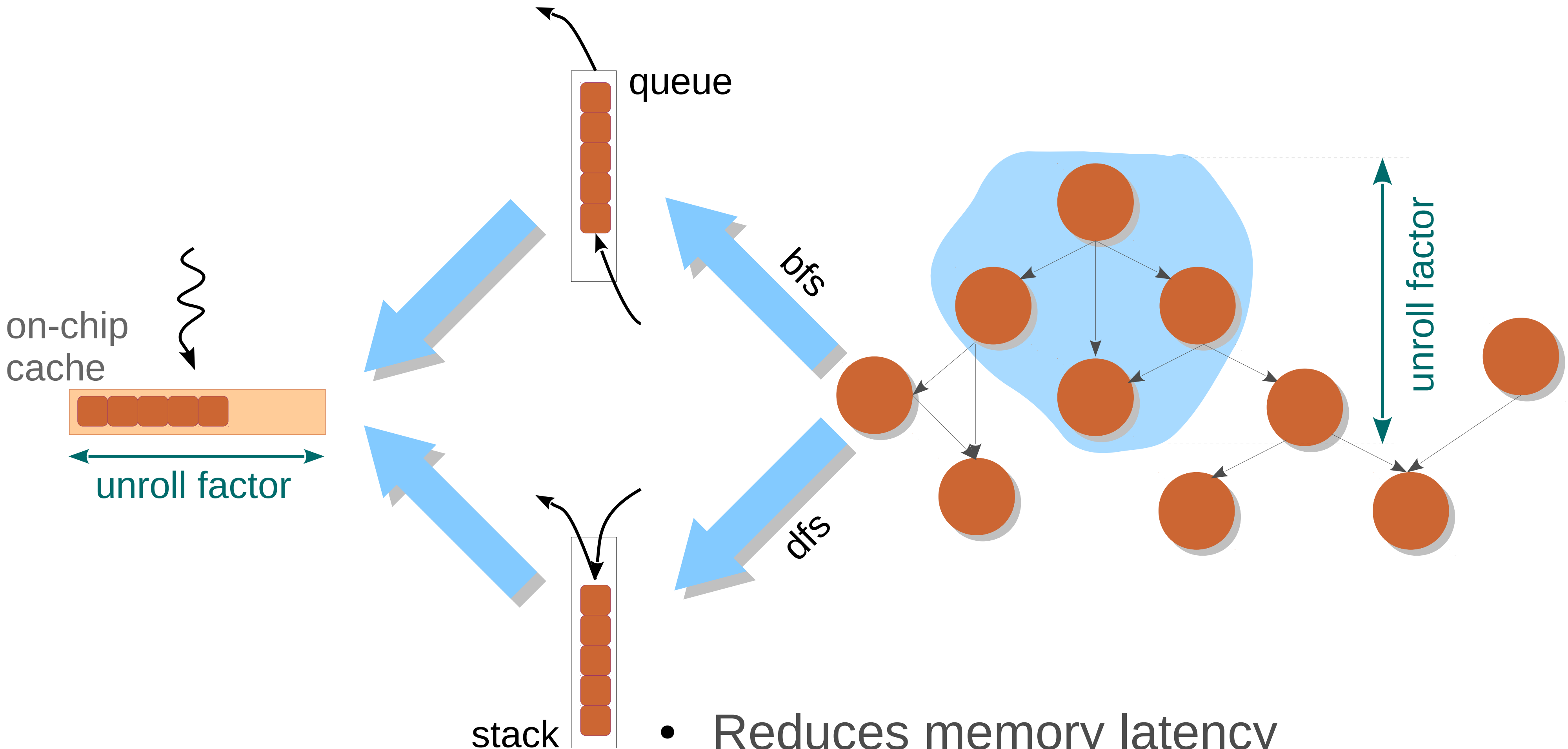
    forall edges (src, dst, wt) {
        $ddst = distance[dst]$
        $altdist = dsrc + wt$

        if $altdist < ddst$
            $distance[dst] = altdist$
    }
}



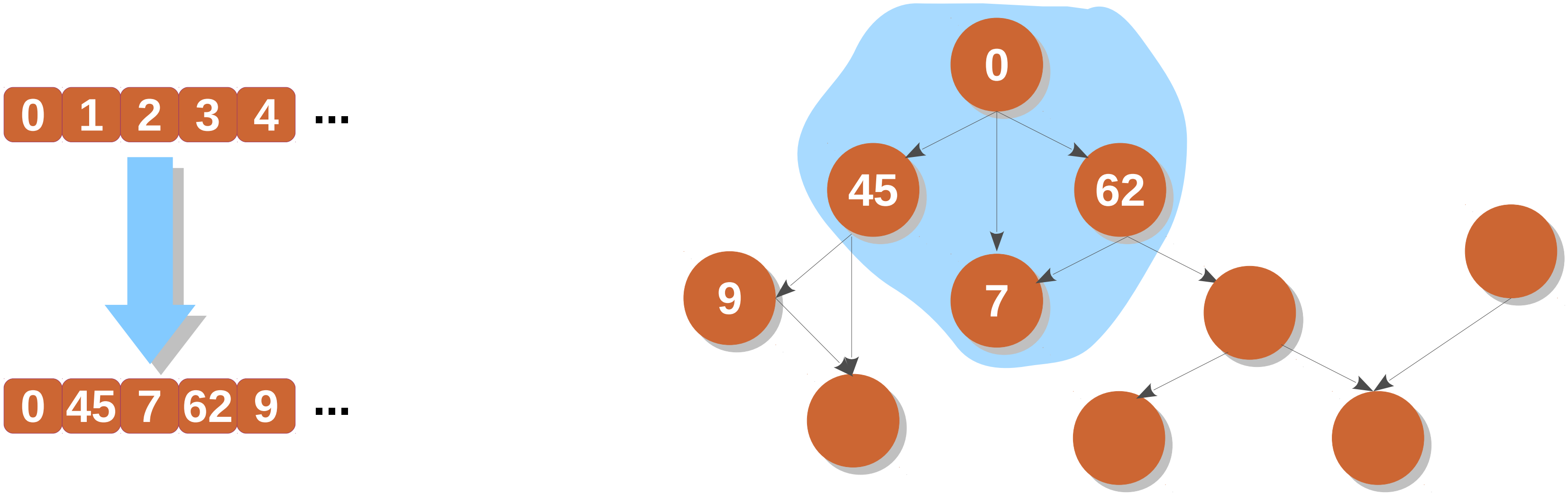
- Improves amount of computation per thread invocation
- Need to ensure absence of races
- Propagates information faster

Topology-driven: Exploiting Memory Hierarchy



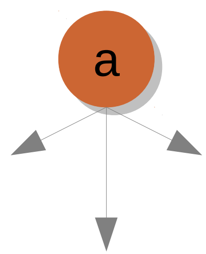
- Reduces memory latency
- Requires careful selection of unroll factor

Topology-driven: Improved Memory Layout

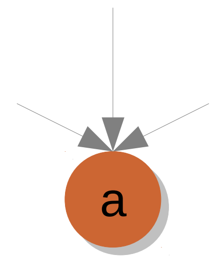


- Bring logically close graph nodes also physically close in memory
- Improves spatial locality

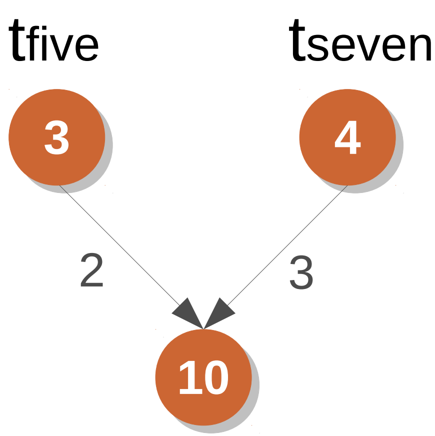
Improving Synchronization



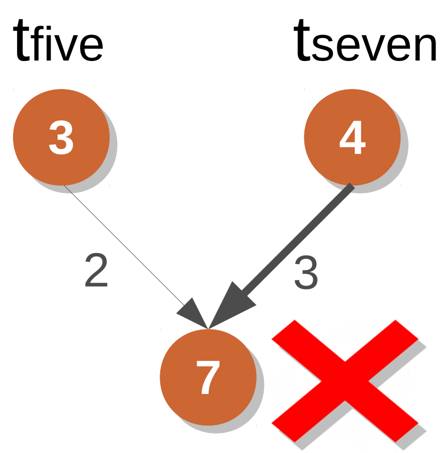
push-based



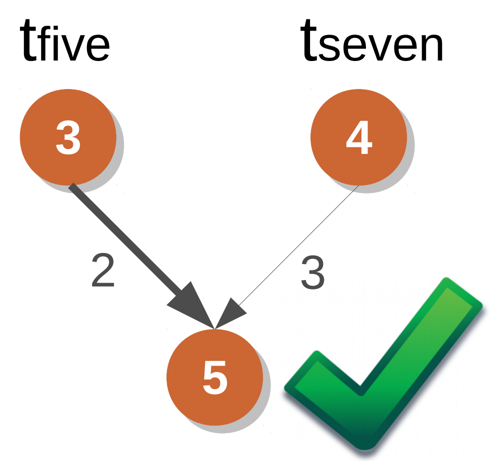
pull-based



Atomic-free update

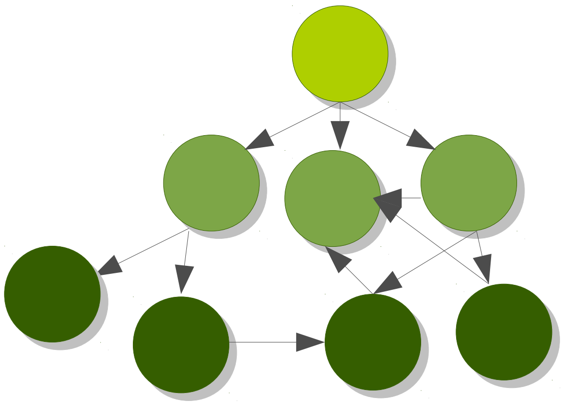


Lost-update problem

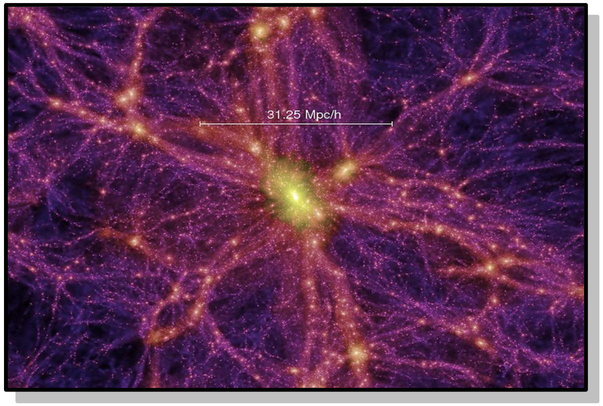


Correction by topology-driven processing, exploiting monotonicity

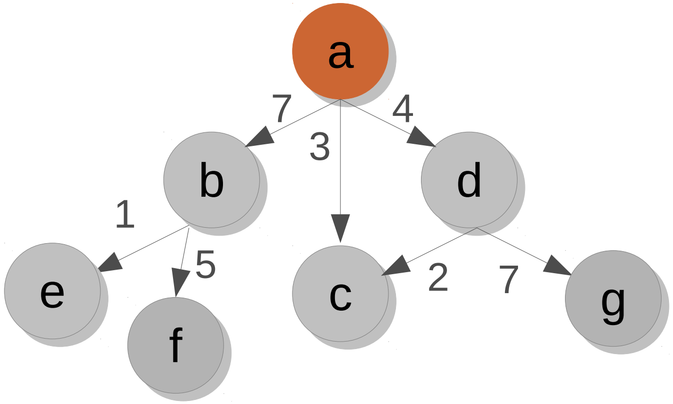
Irregular Algorithms on GPUs



Breadth-first search



Barnes-Hut n-body simulation

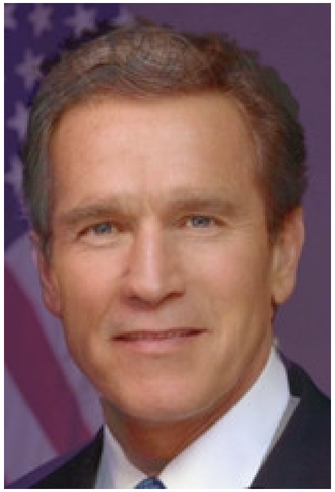


Single-source shortest paths

- Better memory layout
- Kernel unrolling
- Local worklists
- Improved synchronization

Application	Speedup
BFS	48
BH	90
SSSP	45

Identify the Celebrity

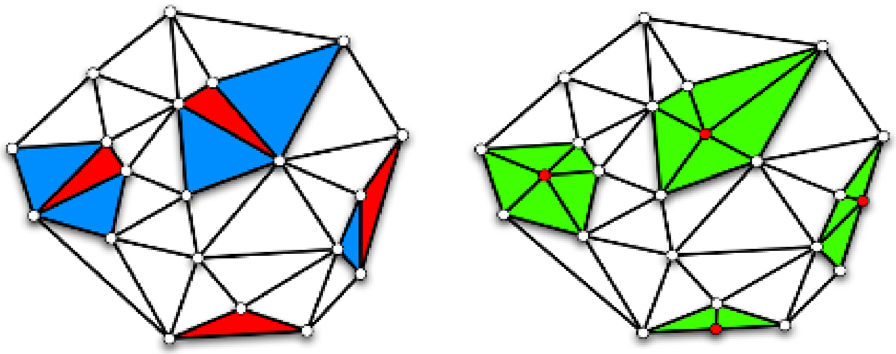


Source: wikipedia

What is a morph?

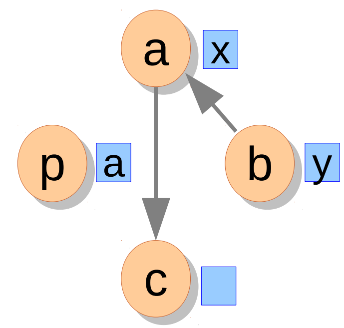
Source: wikipedia

Examples of Morph Algorithms

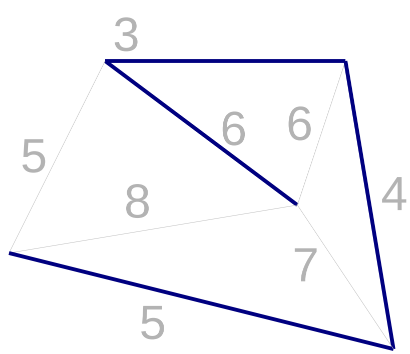


Delaunay Mesh Refinement

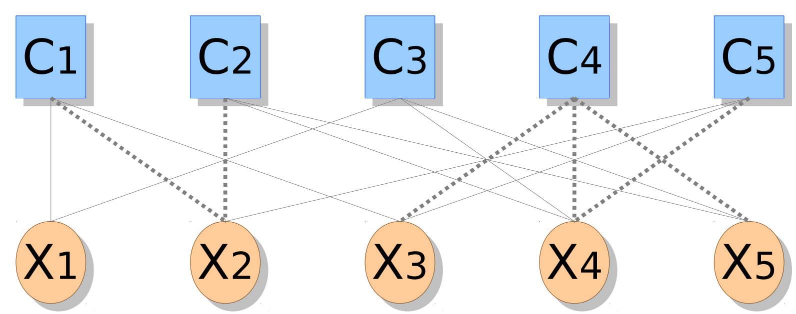
```
a = &x
b = &y
p = &a
*p = b
c = a
```



Points-to Analysis



Minimum Spanning
Tree Computation



Survey Propagation

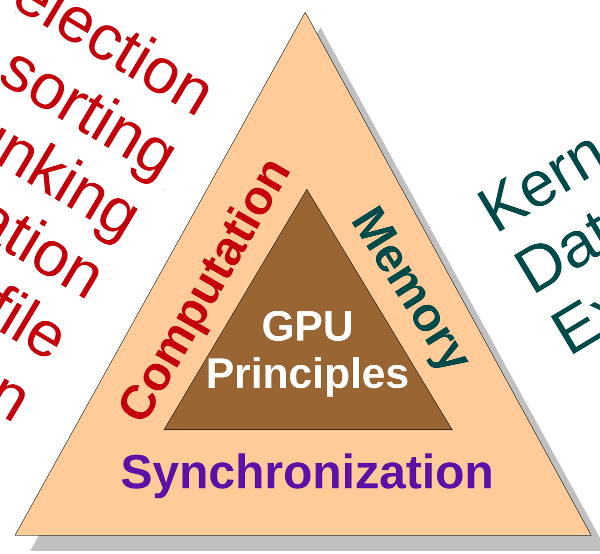
Challenges in Morph Algorithms

- Synchronization
  - locks are prohibitively expensive on GPUs
  - atomic instructions quickly become expensive
- Memory allocation
  - changing graph structure requires new strategies
  - memory requirement cannot be predicted
- Load imbalance
  - different modifications to different parts of the graph
  - work done per node changes dynamically
  - leads to thread-divergence and uncoalesced memory accesses

GPU Optimization Principles

Communication onto computation
Following parallelism profile
Pipelined computation

Algorithm selection
Work sorting
Work chunking



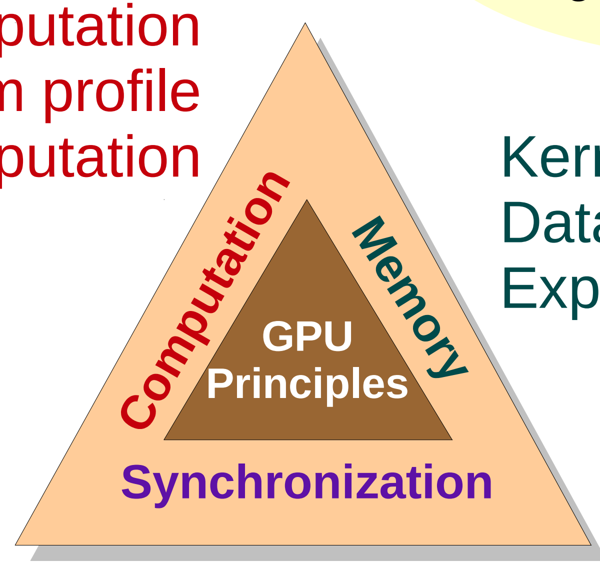
Kernel transformations
Data grouping
Exploiting memory hierarchy

Avoiding synchronization
Coarsening synchronization
Race and resolve mechanism
Combining synchronization

GPU Optimization Principles

Algorithm selection
Work sorting
Work chunking
Communication onto computation
Following parallelism profile
Pipelined computation

These optimization principles are **critical** for high-performing irregular GPU computations.



Kernel transformations
Data grouping
Exploiting memory hierarchy

Avoiding synchronization
Coarsening synchronization
Race and resolve mechanism
Combining synchronization

Approximations

- Reduced execution — Iter. $>K \rightarrow K$
  - reduce the number of iterations
- Partial graph processing — Edge $>K \rightarrow K$
  - process fewer graph elements
- Graph compaction — Vertex $u \rightarrow v$
  - reduce the graph size
- Approximate attribute values — Value $v \rightarrow v / K$
  - reduce the number of distinct values
- ...

Approximation A(Domain D, Function F)
Function F: entity → entity
entity belongs to Domain D.

Synchronization Saurabh, Ganesh	**Energy** Jyothi Krishna, Nikitha	**Approximations** Somesh, Tejas
Graph DSL Shashidhar	**Clustering** Anju	**Testing and Android** Shouvick, Aman
Autoparallelizers Prema	**Community Detection** Akash, Srivatsan	**Imaging** Mullai
Steiner Trees Rajesh	**Consistency** Diptanshu	**Pravin** Hydrodynamics

Gajendra

- Invited paper at ACM Transactions on Parallel Computing
- Ranganath research award at IIT Madras in 2019
- Winner of HiPC Parallel Programming Challenge: Intel track in 2017
- Distinguished Paper Award at PPoPP 2016
- Best Paper Award at HiPC Student Research Symposium 2015
- ...

Exercises

- Find if true dependence exists for the loop.

```
for (ii = 0; ii < 10; ++ii) {
    a[2 * ii] = ... a[ii + 1] ...
    a[3 + ii] = ... a[5 * ii] ...
}
```

- Represent a graph as adjacency list on GPU.
- Represent an input graph in CSR format, and then convert it into a COO format.
- Write a kernel to count degrees of various vertices. Check finally that the sum equals the number of edges.
- Implement shortest path algorithm. Check your implementation against that in CUDA SDK.