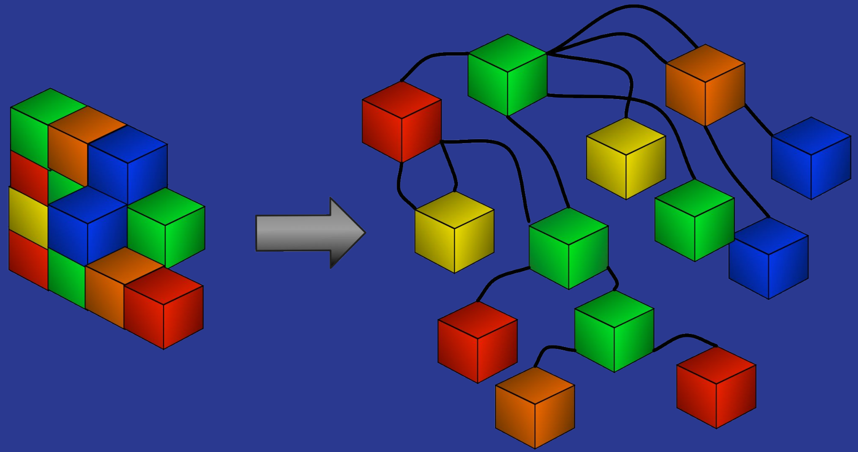
Breadth First Search Optimization

Nick Boukis & Ethan Wayda

Graph Transversal

Definitions and Breakdown



Graph Transversal

Given Graph | Find Connecting Nodes | All Nodes are Visited

Graph Representation

- Edge List
- Adjacency List
- Adjacency Matrix

Our Choices

- Adjacency Matrix
- Undirected Graph

Parent Array

- Keeps track of path taken
- Output that can be interpreted and understood

Visited Array

- Keeps track of which nodes are visited
- Avoids cycles
- Provides end case
- O(V+E)

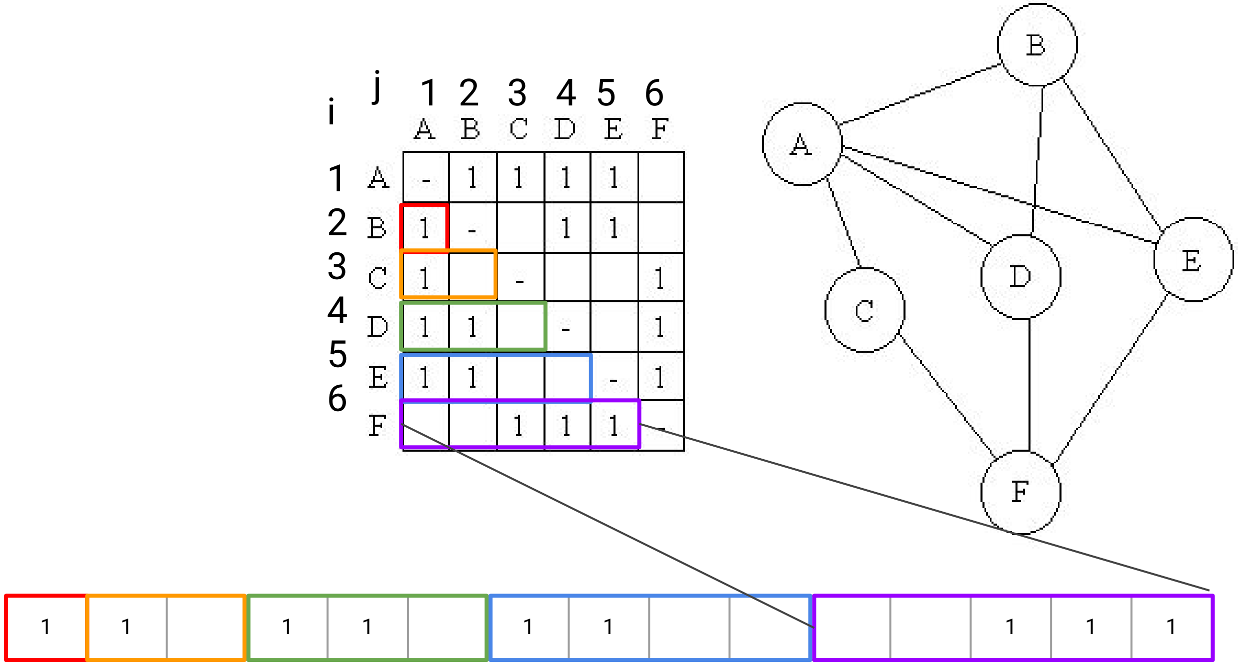
Undirected Graph

e redundant

ut half the

$$\frac{N-1}{2N} \approx .5$$

$$\frac{i(i-1)}{2} + j$$



Breadth First Search

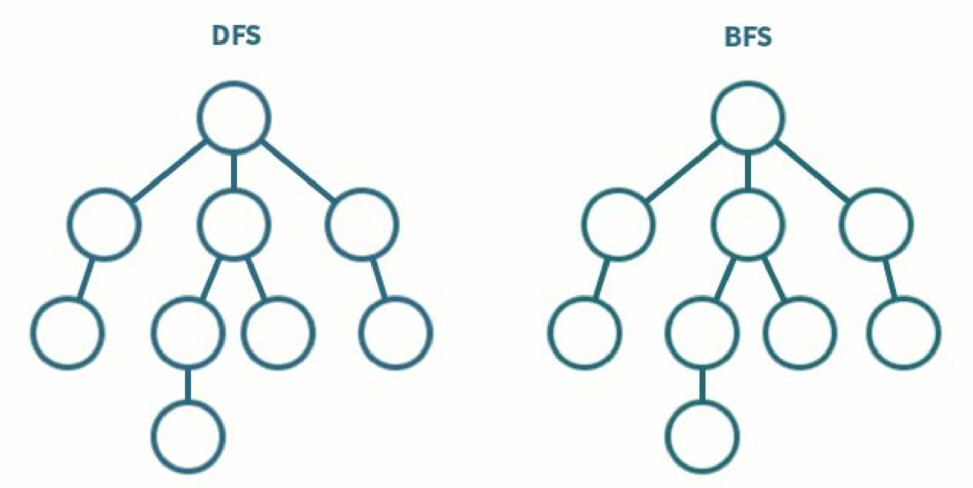
Logic and Concepts

Definition

Graph Transversal

A method of traversing a tree/graph where the neighbors of a given node must be explored before visiting more nodes.

Problem - How do you maintain order?



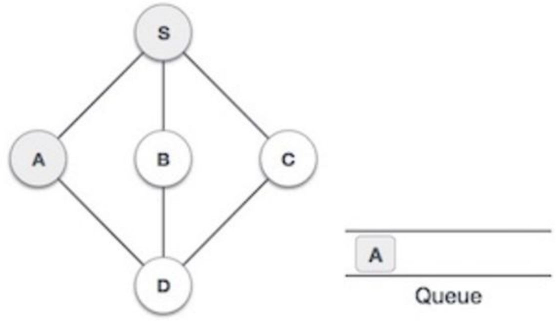
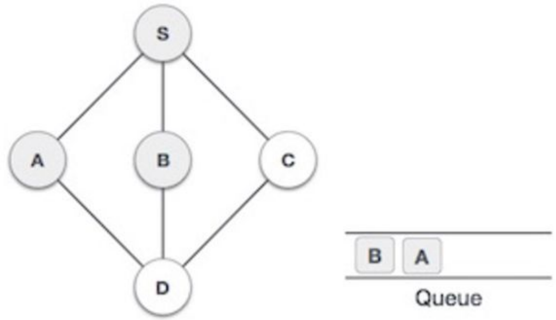
Simple Approach

Queue

When a node is visited add it to the queue

After all neighbors are checked pop the next node off the graph

Repeat

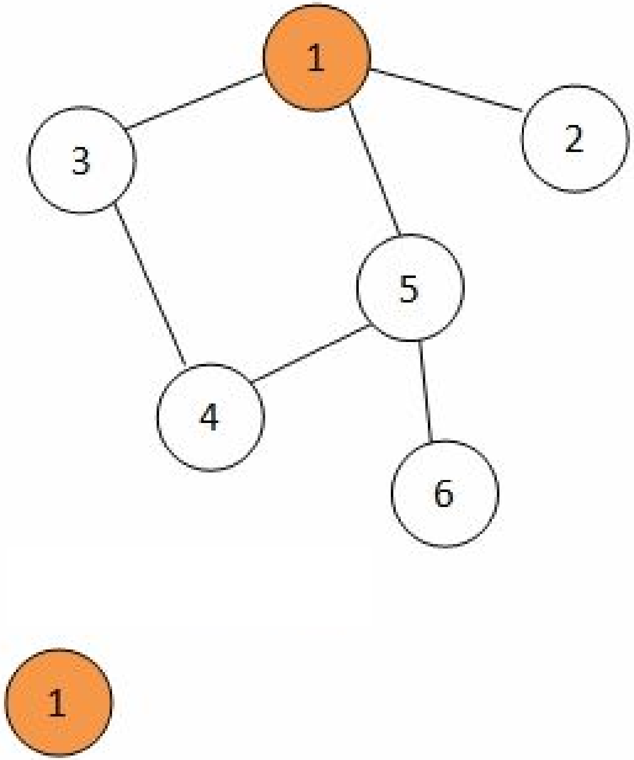
3.		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4.		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.

Memory Access

Large
$O(V^2)$
Multiple arrays to access
(Parents, Visited, Graph, Queue)

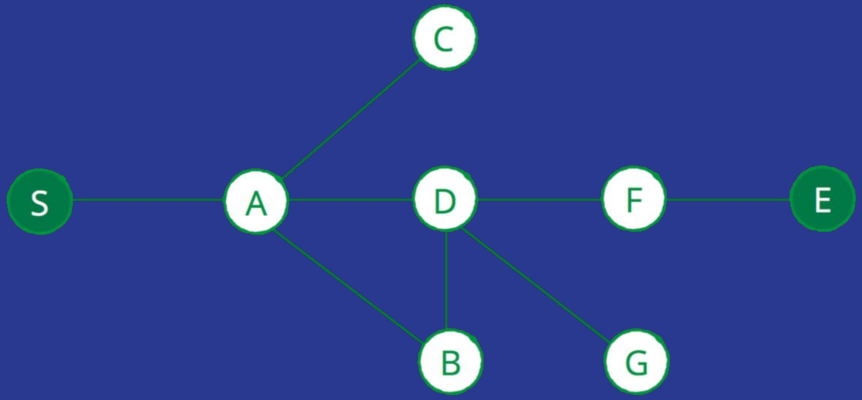
Random
No ordered access

Overall
Memory Access is our biggest constraint



Hybrid Approach

Exploring the Frontier



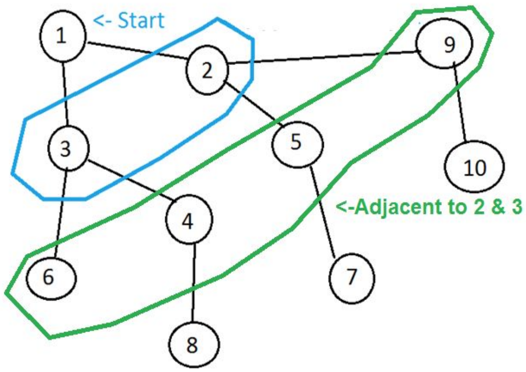
Hybrid Approach - Frontier

Definition

Set of vertices that are the same distance from the root node

Current frontier is updated through one-step intervals throughout the graph

Visualization



Advantage

Creates groupings of nodes based on breadth

Allows multiple nodes to be analyzed at a time

Enables parallelization and perspective of the graph

Hybrid Approach - Top2Down

- Similar to the simple method
- In = Array of the current frontier
- Out = Array of the next frontier

Loops through the current frontier and then loops through each of its neighbors

If the neighbor is not visited, update its information and add it to the next frontier

Ideally a small current frontier

Algorithm 2: Top2down-step(in, out, vis, parents)

```
for v ∈ in do
   for n ∈ neighbors[v] do
      if n ∉ vis then
         parents[n] ← v
         out ← out ∪ {n}
         vis ← vis ∪ {n}
      endif
   endfor
endfor
```

Hybrid Approach - Down2Top

Approach for when the frontier is large

Analyzes nodes that are not on the current frontier and attempts to find their parents

Shortens neighbor calls, because it only needs to find one parent

Algorithm 3: Down2top-step(*in, out, vis, parents*)

```
for v ∉ vis do
    for n ∈ neighbors[v] do
        if n ∉ in then
            parents[v] ← n
            out ← out ∪ {n}
            vis ← vis ∪ {n}
        endif
        break
    endfor
endfor
```

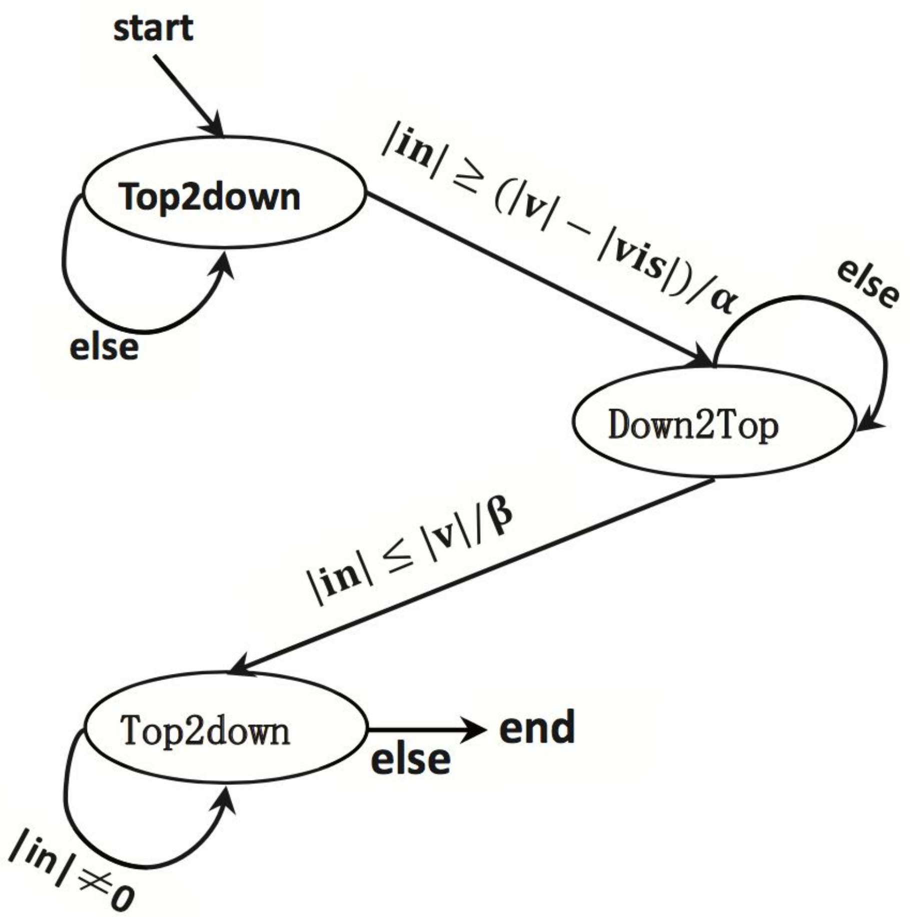
Hybrid Approach

Algorithm 1: Level-synchronized BFS

```
Input  : G(V,E), root // Graph vertex
Output : p // predecessor map
in ← {root}
vis ← {root}
p[root] ← r
while in ≠ Φ do
    out ← Φ
    one-level-step (in, out, vis, p )
    swap (in, out )
```

One-level-step:

- Top2down
- Down2top

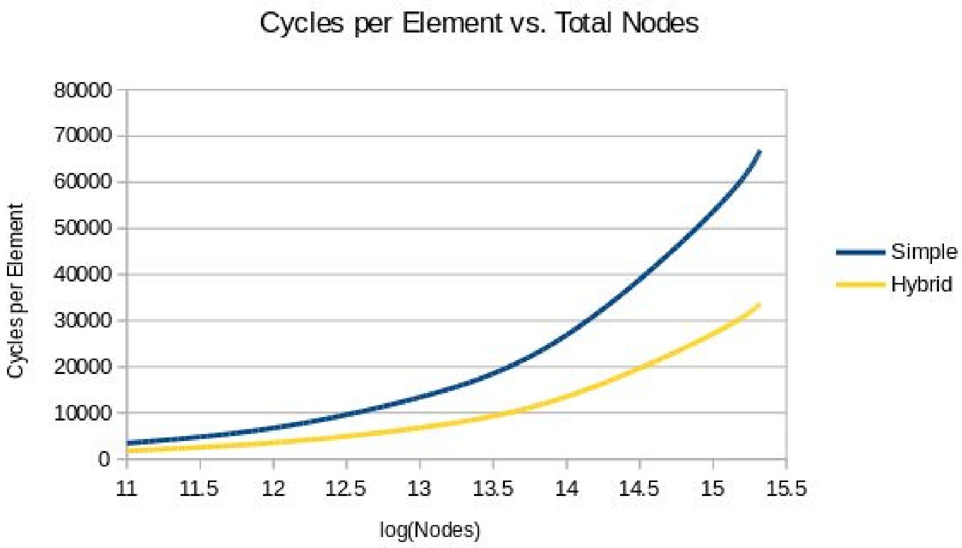


Simple vs Hybrid

Hybrid BFS achieves speedup of ~**48.88%** relative to Simple BFS

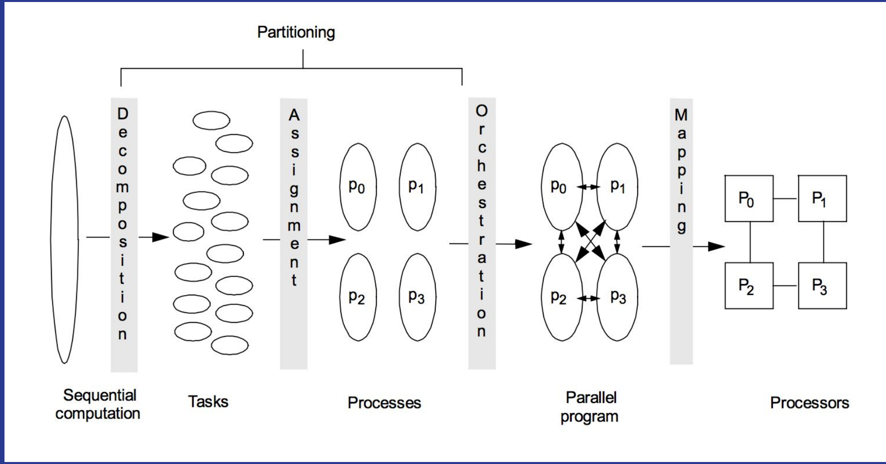
Better handling of the frontier

Overall decrease of internal looping



Parallelization

Logic and Concepts



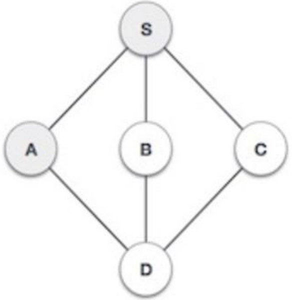
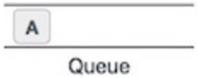
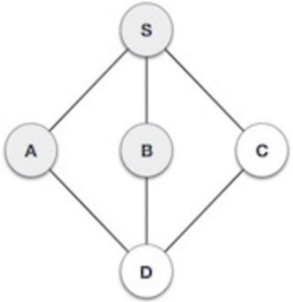
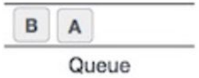
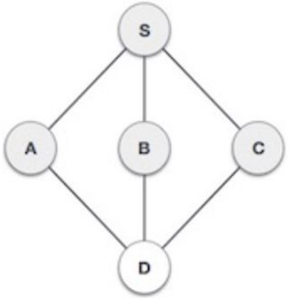
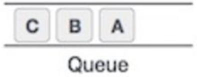
Simple Parallel Approach

All threads share same queue head

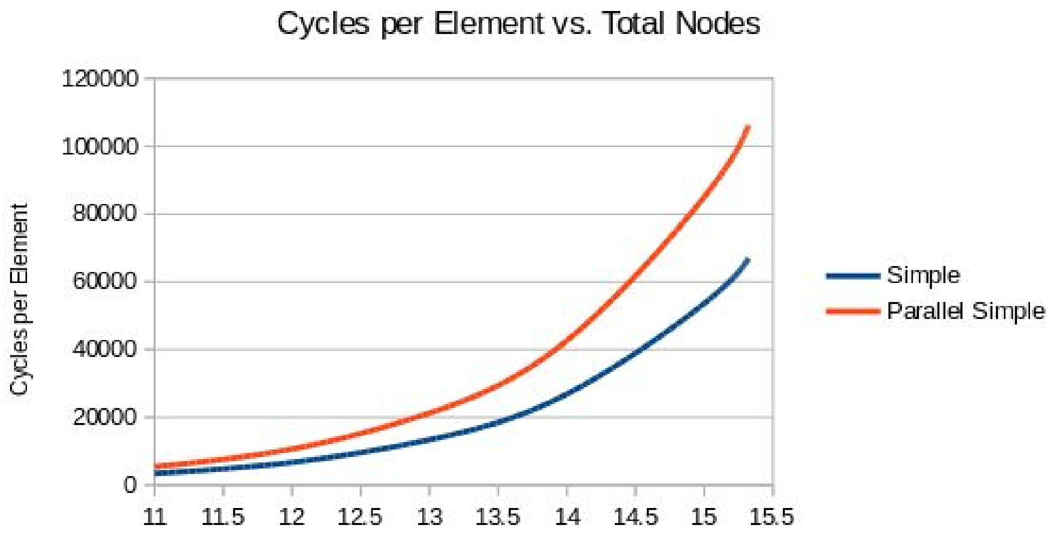
Threads split up which neighbors to visit

Each threads waits for the rest before checking the next value in the queue

Queue is locked

3.		Thread 0 Finds A 	We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4.		Thread 1 Finds B 	Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.
5.		Thread 2 Finds C 	Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

Simple Parallel Approach Results

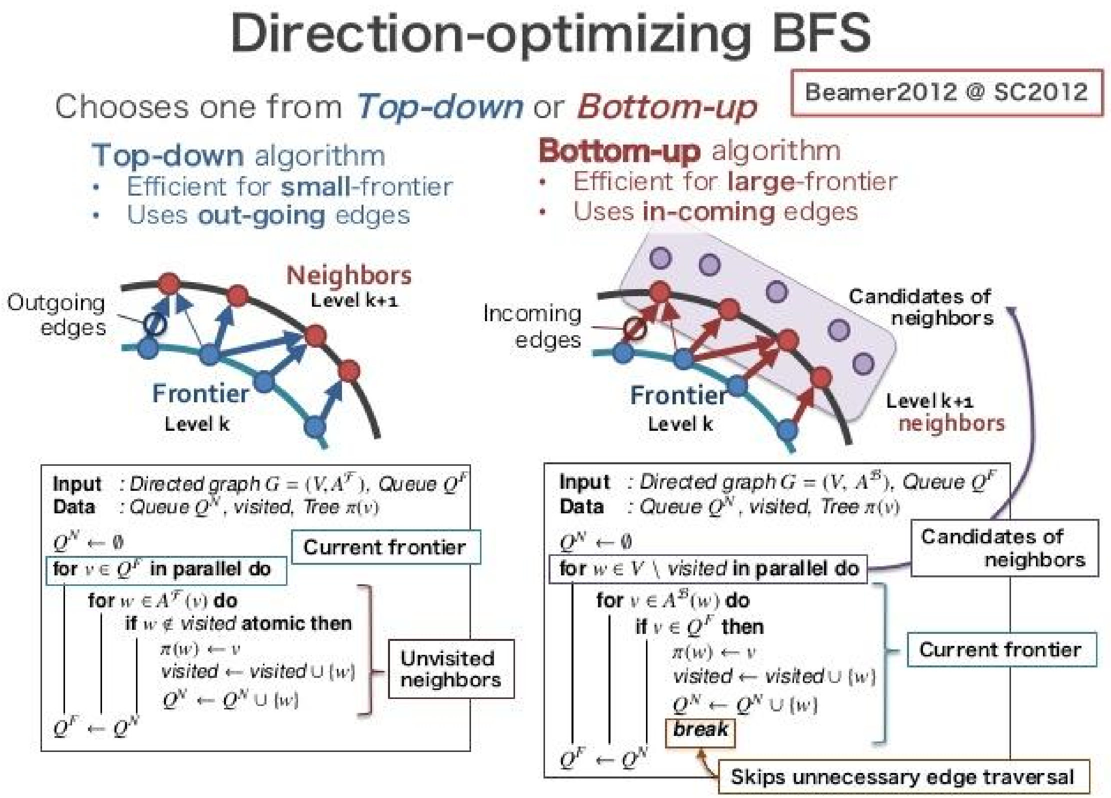


Parallel approach led to 1.587x slower CPE than serial

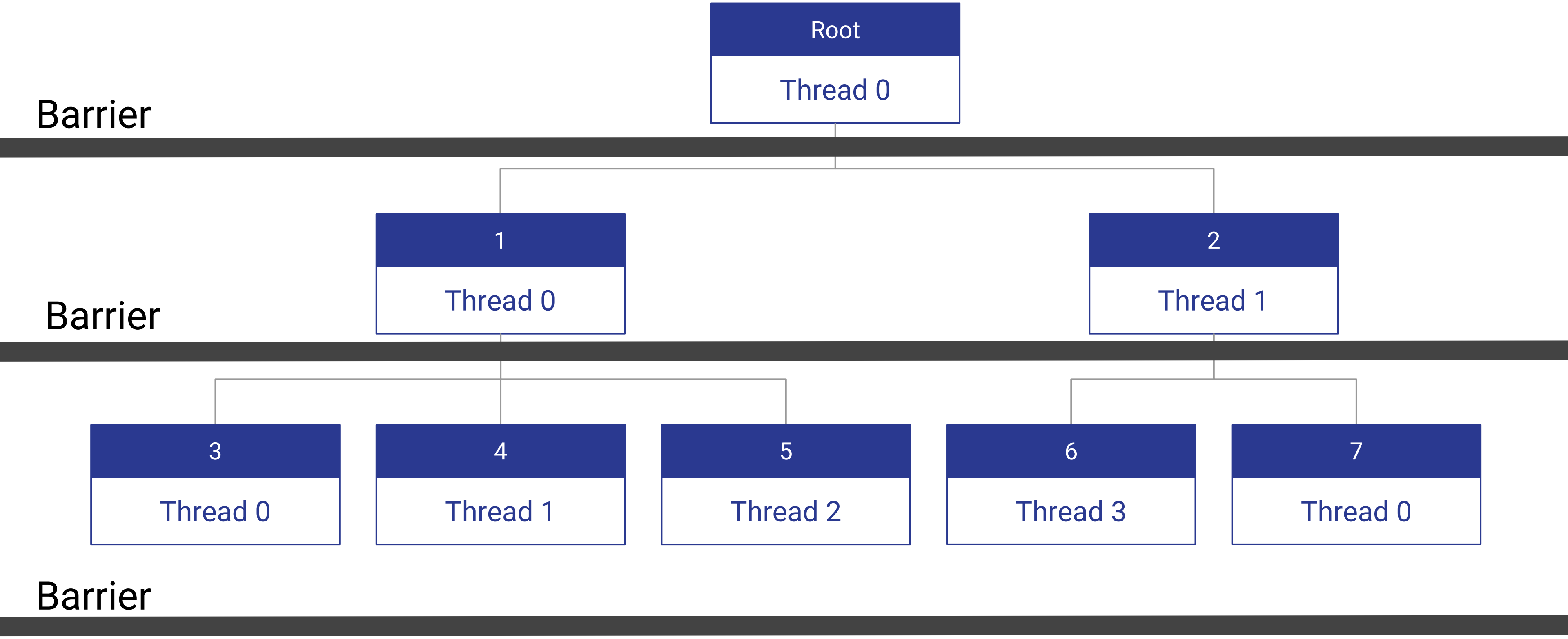
Poor sharing of the queue

Threads spend more time waiting than doing work

Hybrid Parallel Approach



The Approach: Top-Down



Data

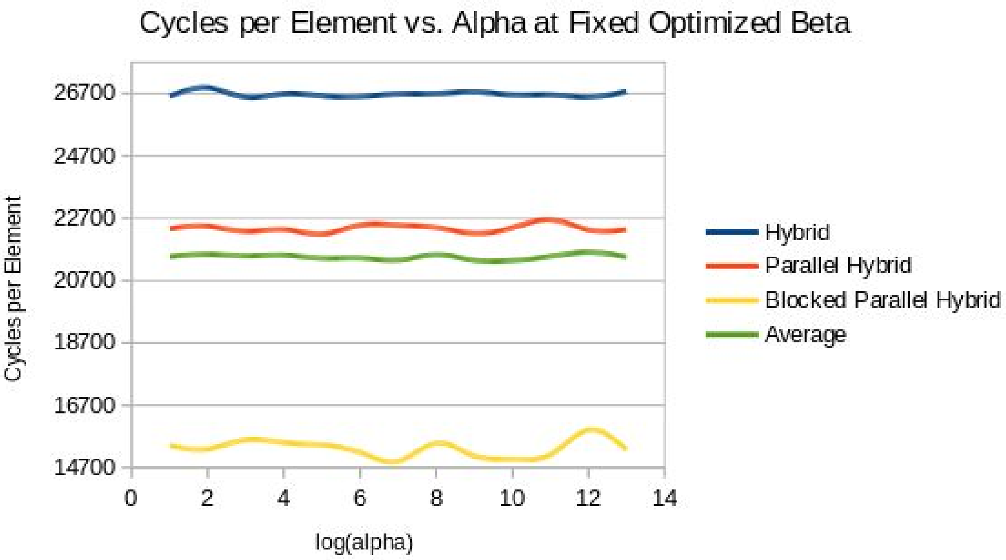
Alpha (α)

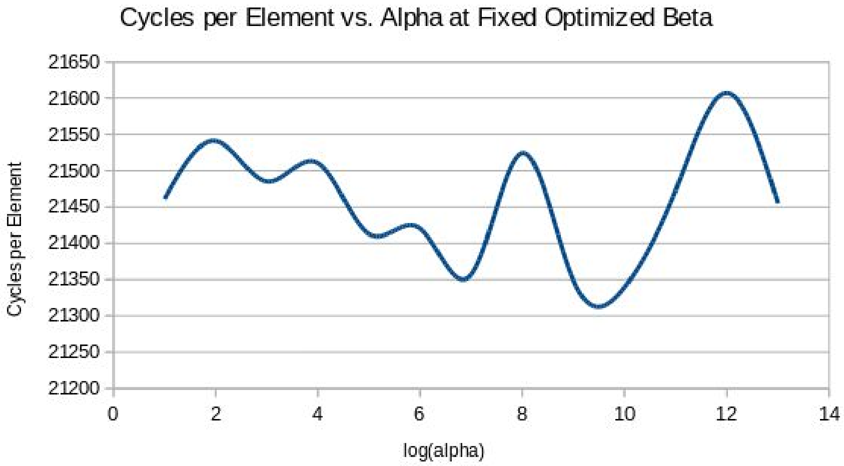
Decides when the frontier is too large

1024 for 32k nodes

Optimal:

$$\alpha = \frac{N}{32}$$





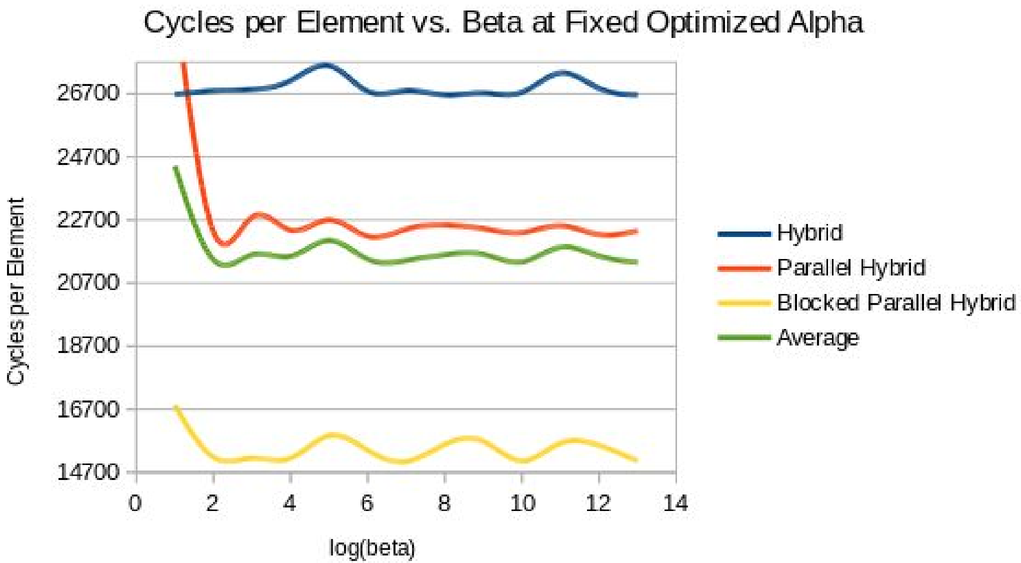
Beta (β)

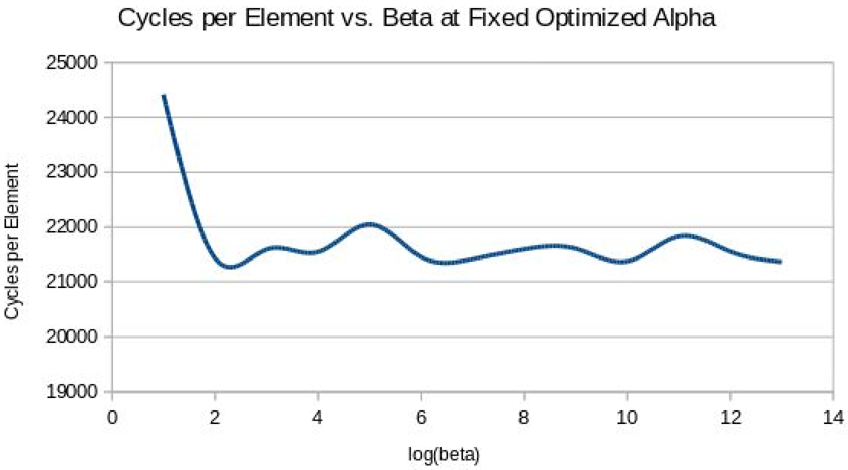
When the frontier is an appropriate size again

16 for 32k nodes

Optimal:

$$\beta = \frac{N}{2048}$$





Blocked

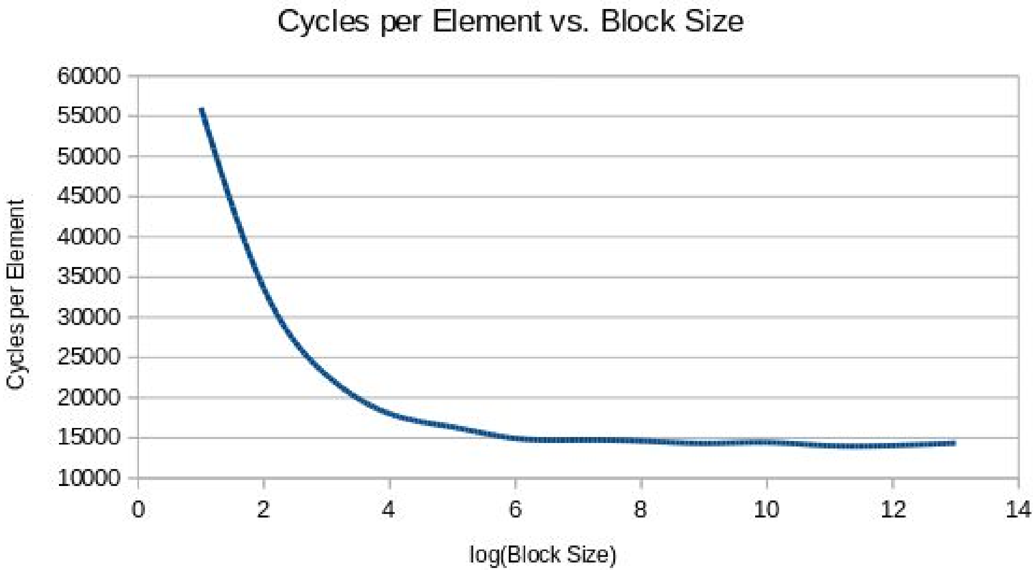
- Temporal locality
- 2048 for 32k nodes

- Optimal:

$$Block\ Size = \frac{N}{16}$$

- Lower bound:

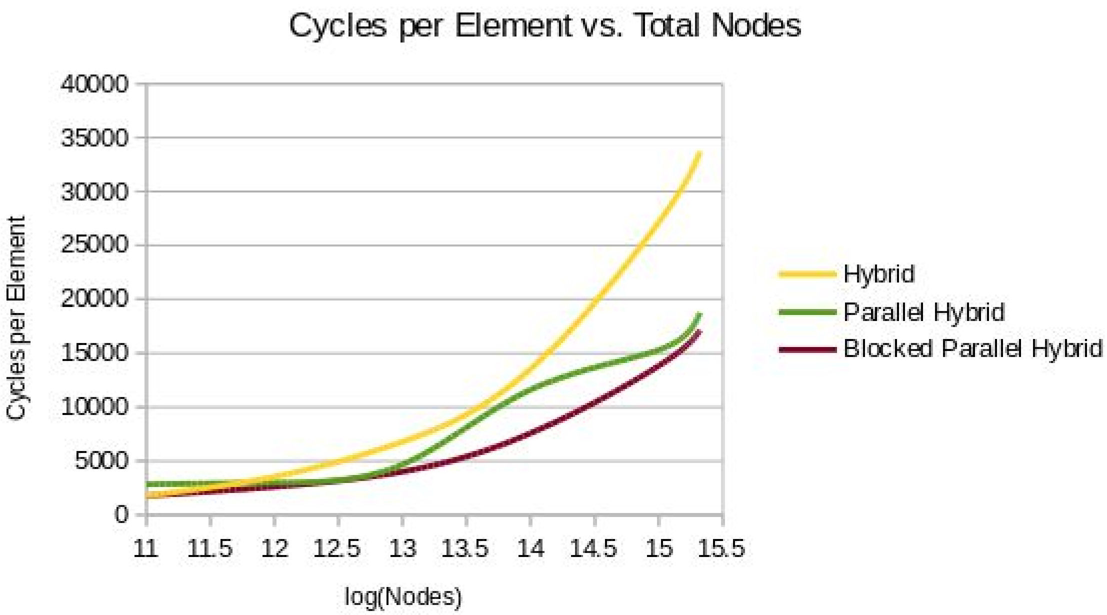
$$Block\ Size > \frac{N}{512}$$



Parallel Hybrid Results

Method	Speedup*
Parallel Hybrid	43.93%
Blocked Parallel Hybrid	49.05%

** relative to Serial Hybrid*



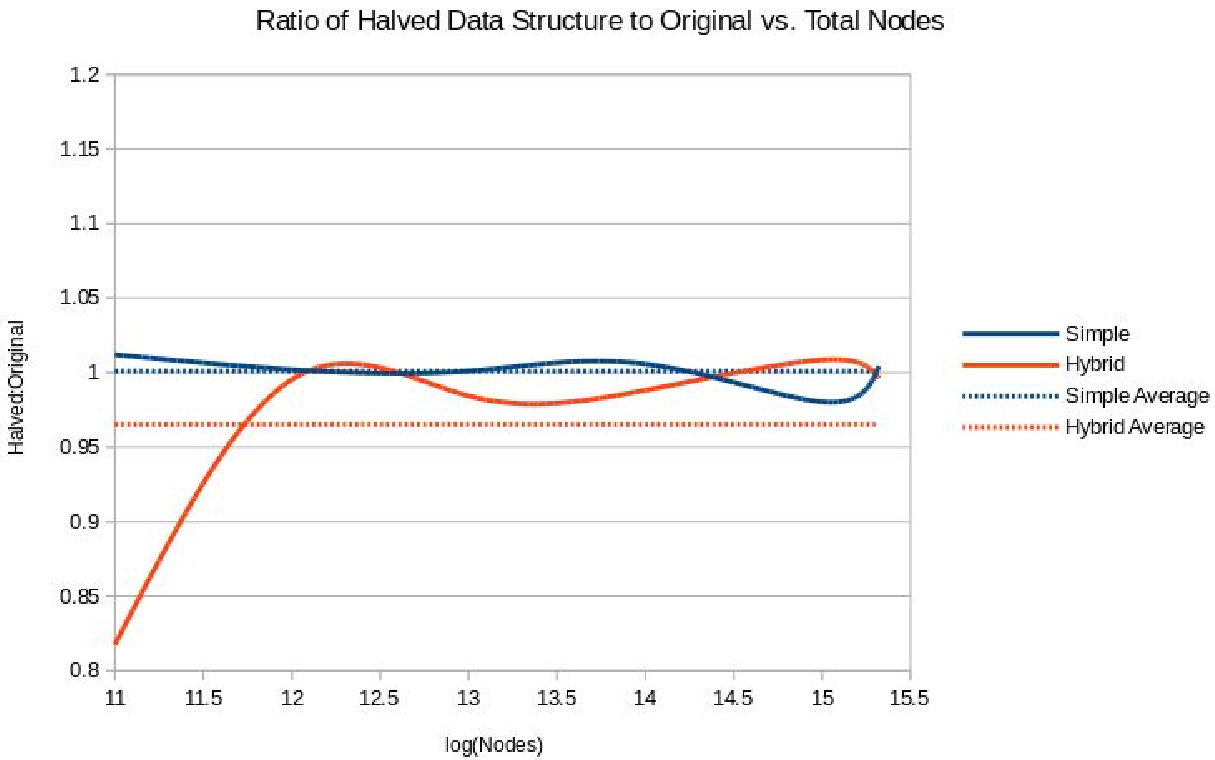
Multiple threads allow faster traversing of the frontier

Still need to sync up between steps

Halved Adjacency Matrix vs Original

Method	Halved:Original
Simple	1.001
Hybrid	0.965

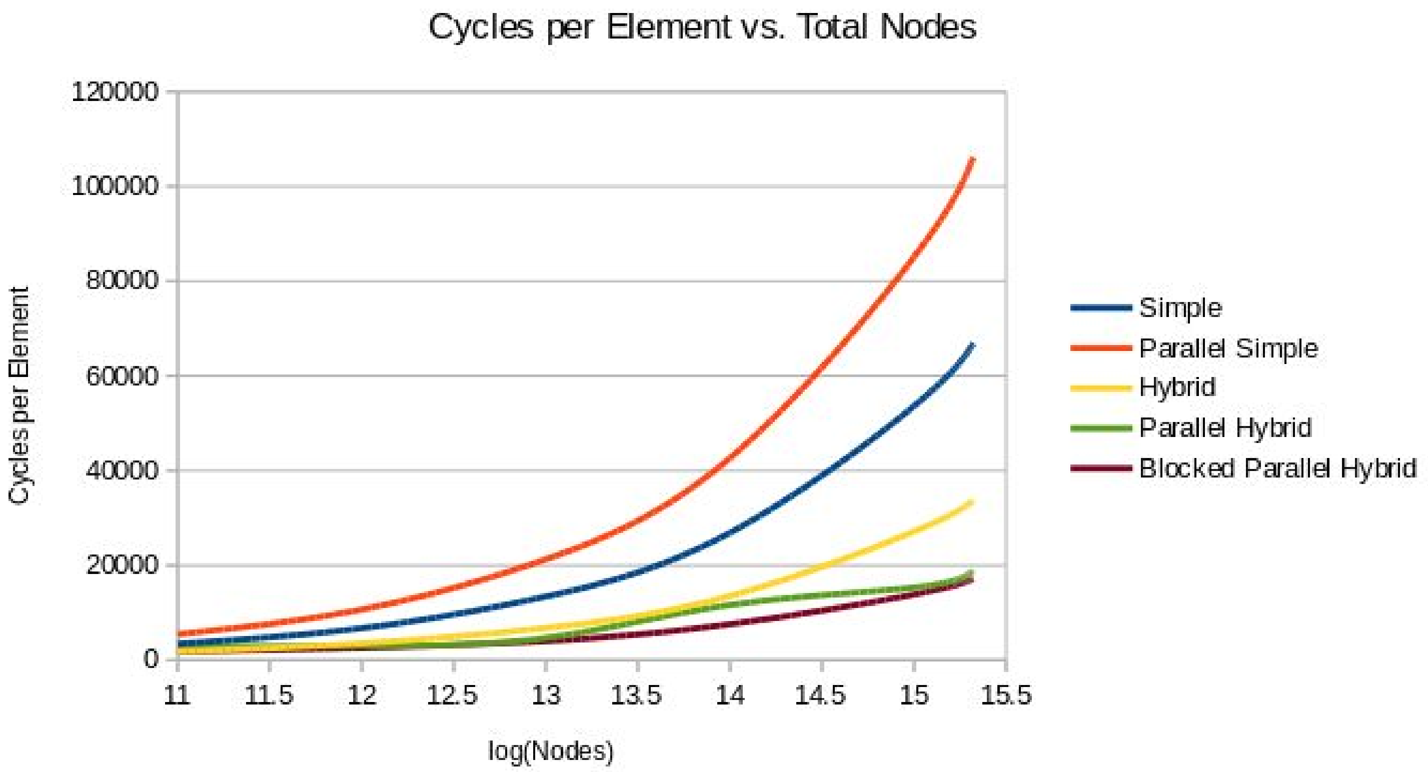
Reduces redundancy in Down2top() method



Parallel Comparison Summary

Method 1	Method 2	Speedup
Serial Hybrid	*Serial Simple*	48.88%
Parallel Simple	*Serial Simple*	- 58.70%
Parallel Hybrid	*Serial Hybrid*	43.93%
Blocked Parallel Hybrid	*Serial Hybrid*	49.05%
Blocked Parallel Hybrid	*Serial Simple*	74.28%

Parallel Comparison Summary



Questions?