Breadth First Search Optimization

High Performance Programming with
Multicore & GPUs

By Chen-Yu Chang

Graph Traversal

Definition

Graph traversal refers to the process of visiting each vertex in a graph. Such traversals are classified by the order in which the vertices are visited.

Procedure

Graph

- ➔ Adjacency List
- ➔ Adjacency Matrix
- ➔ Undirected Graph

Find Connections

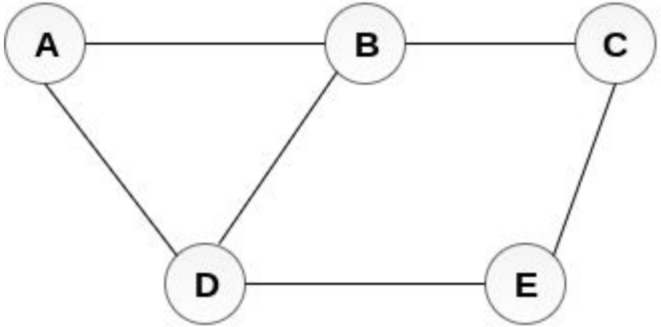
- ➔ Match from parent nodes

Check Visited

- ➔ Record when the node is visited
- ➔ Prevent from repeating

How Fast: O(|V|+|E|) (Worst Case) where V (vertices), E (Edges)

Graph Traversal



Undirected Graph

$$
\begin{array}{c|ccccc}
 & A & B & C & D & E \\
\hline
A & 0 & 1 & 0 & 1 & 0 \\
B & 1 & 0 & 1 & 1 & 0 \\
C & 0 & 1 & 0 & 0 & 1 \\
D & 1 & 1 & 0 & 0 & 1 \\
E & 0 & 0 & 1 & 1 & 0
\end{array}
$$

Adjacency Matrix

Image Source: https://www.javatpoint.com/graph-representation

Breadth First Search

Definition

BFS is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

Compare to DFS

Unlike BFS, DFS explores the node branch as far as possible before being forced to backtrack and expand other nodes.

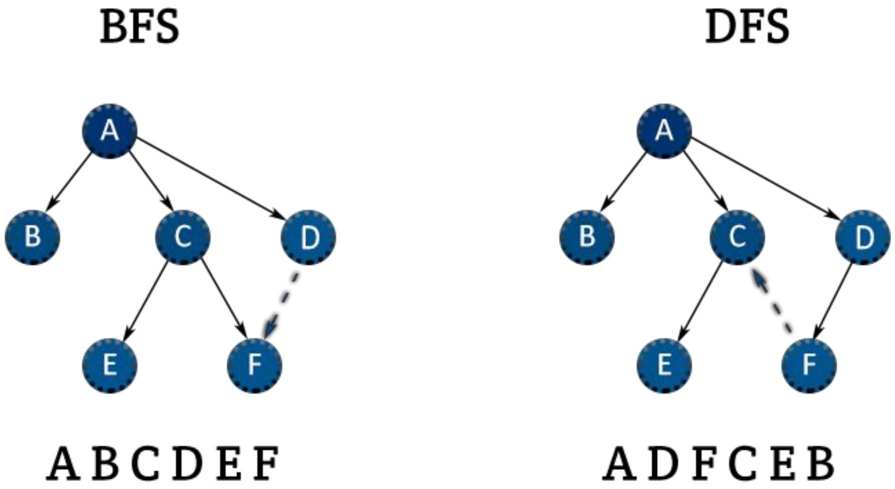
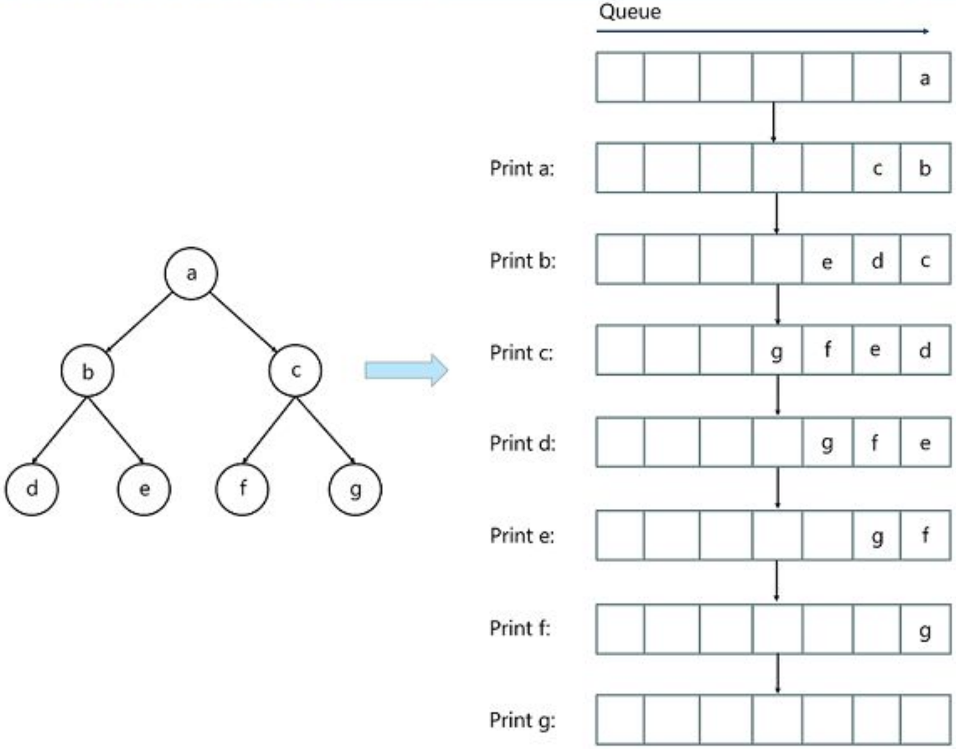


Image Source: https://open4tech.com/bfs-vs-dfs/

Serial Method

Method: Queue

Key: When the node is visited, add to the queue. After going to the next level, dequeue the previous node and add the new ones



							a	
Print a:						c	b	Print 'a' & insert its child nodes into the queue
Print b:					e	d	c	Print 'b' & insert its child nodes into the queue
Print c:				g	f	e	d	Print 'c' & insert its child nodes into the queue
Print d:					g	f	e	Print 'd' & insert its child nodes into the queue
Print e:						g	f	Print 'e' & insert its child nodes into the queue
Print f:							g	Print 'f' & insert its child nodes into the queue
Print g:								Print 'g' & insert its child nodes into the queue

Image Source: https://www.edureka.co/blog/breadth-first-search-algorithm/

Challenges

- Need to set a high capacity for he queue if we operate BFS for huge graph
- In huge graph, BFS will operate slowly since all the other levels need to wait for the previous levels
- Memory access is another constraint : $O(V^2)$
  - Including Parents, Visited, Queue, and Graph arrays

Hybrid Algorithm

Definition

Set the targeted node and find the nodes that are adjacent (same distance to the root node). Group them based on breadth.

Advantage

★ Instead of node by node, the whole algorithm categorize the nodes into groups

★ Potential improvement can be operating multiple nodes at the same time

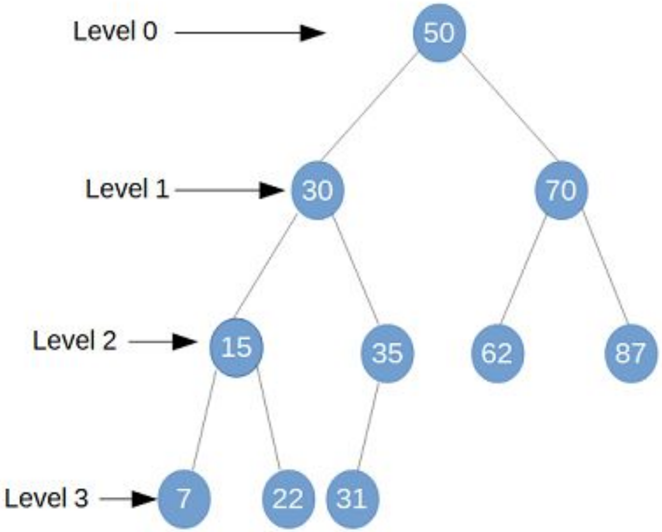
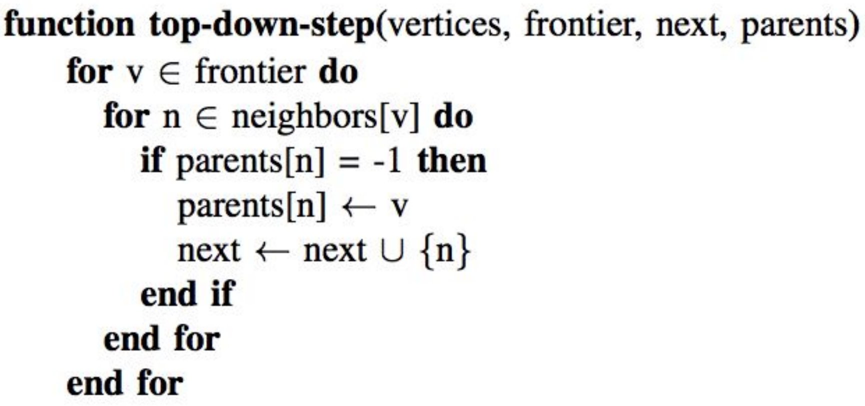


Image Source: https://www.netjstech.com/2019/03/binary-tree-traversal-breadth-first-serach-link-order-java.html

Top-Down Method: O(E)

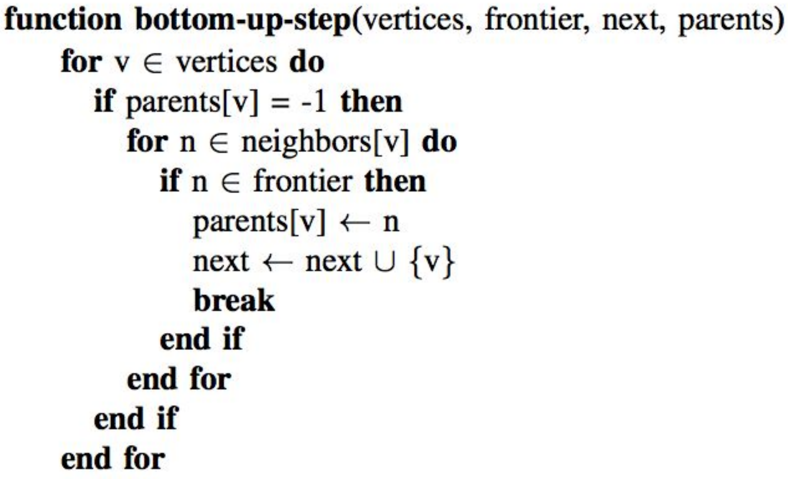


```
function top-down-step(vertices, frontier, next, parents)
    for v ∈ frontier do
        for n ∈ neighbors[v] do
            if parents[n] = -1 then
                parents[n] ← v
                next ← next ∪ {n}
            end if
        end for
    end for
```

- ➔ Efficient on small frontier
- ➔ Loop the current frontier and its neighbors
- ➔ If not visited, add to the next frontier

Image Source: http://www.scottbeamer.net/pubs/beamer-sc2012.pdf

Down-Top Method: O(NV+E)

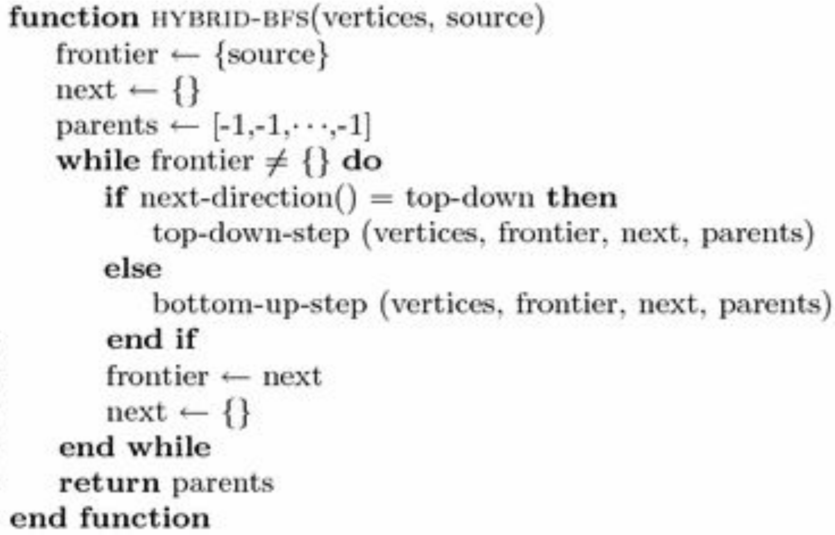


```
function bottom-up-step(vertices, frontier, next, parents)
    for v ∈ vertices do
        if parents[v] = -1 then
            for n ∈ neighbors[v] do
                if n ∈ frontier then
                    parents[v] ← n
                    next ← next ∪ {v}
                    break
                end if
            end for
        end if
    end for
```

- ➔ Efficient on large frontier
- ➔ Find the nodes’ parents that are not in the current frontier

Image Source: http://www.scottbeamer.net/pubs/beamer-sc2012.pdf

Hybrid



```
function HYBRID-BFS(vertices, source)
    frontier $\leftarrow$ {source}
    next $\leftarrow$ {}
    parents $\leftarrow$ [-1,-1,$\cdots$,-1]
    while frontier $\neq$ {} do
        if next-direction() = top-down then
            top-down-step (vertices, frontier, next, parents)
        else
            bottom-up-step (vertices, frontier, next, parents)
        end if
        frontier $\leftarrow$ next
        next $\leftarrow$ {}
    end while
    return parents
end function
```

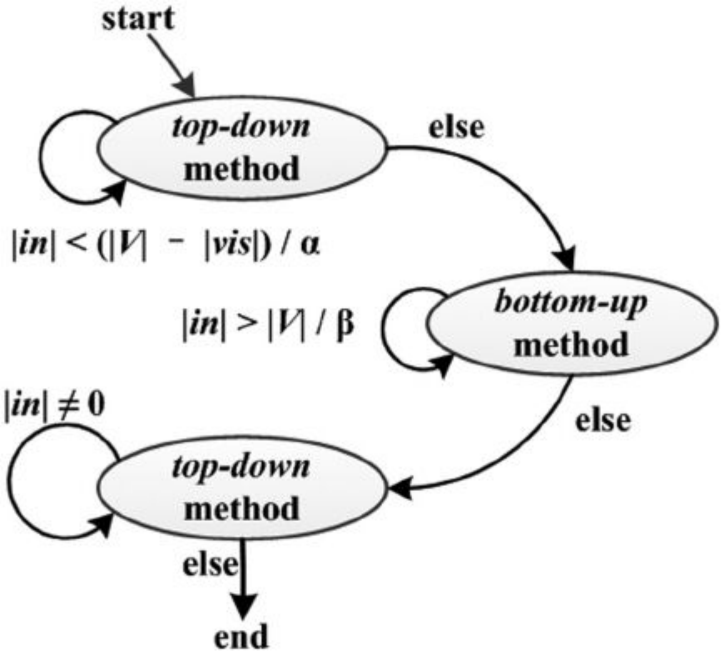


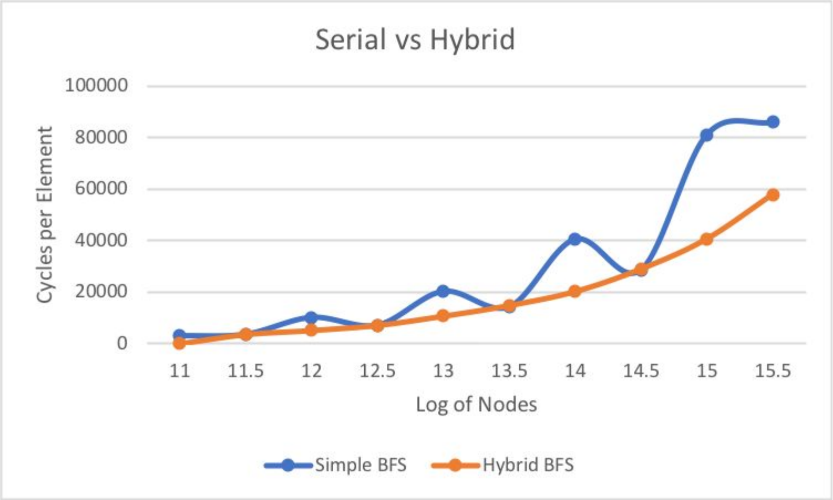
Image Source: https://link.springer.com/article/10.1007/s41019-016-0024-y

Image Source: https://www.researchgate.net/figure/The-state-machine-of-the-hybrid-BFS-algorithm_fig1_260824238

Data Comparison

By comparing the cycles per element, we gain the data from using the serial code & hybrid code.

➔ Overall, the hybrid approach seems to decrease the iterations and shows a better control on frontiers



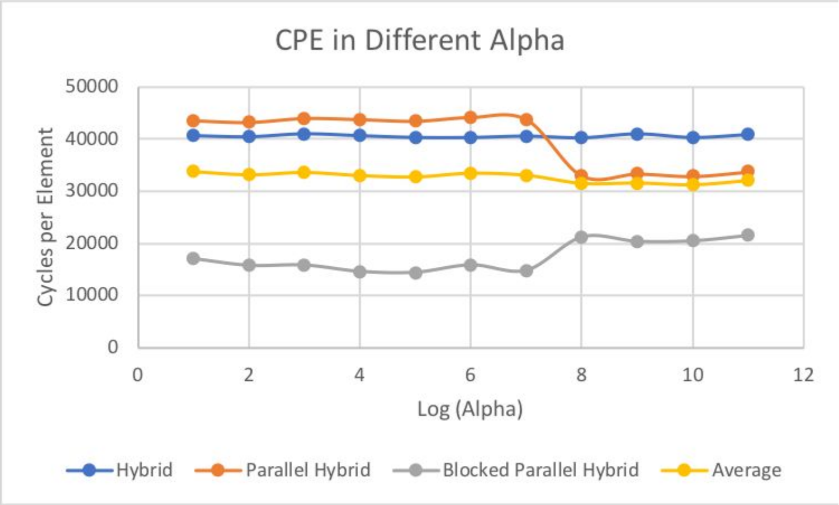
α (Alpha)

Determine when the frontier is too large

From the data:

- ➔ When there are 32768 nodes, the optimized Alpha is at about 1024.
- ➔ It also shows the logical result in the graph on the right.

Usage: Compensate for bottom-up finishing



β (Beta)

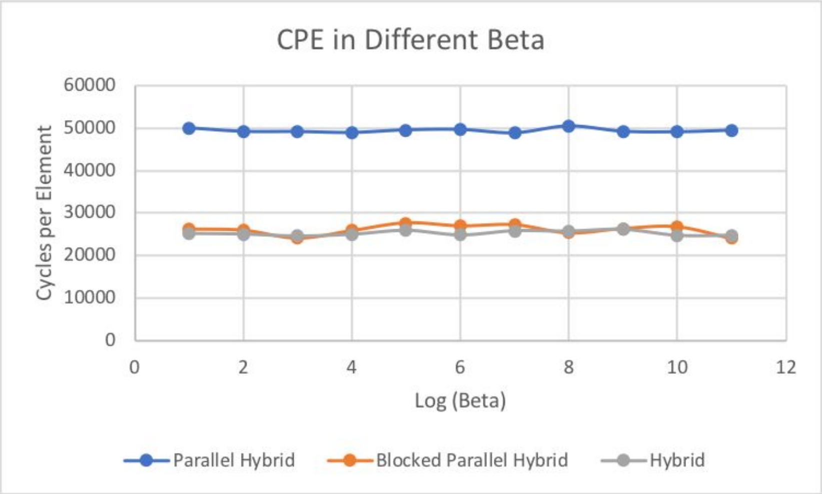
Less important comparing to alpha

Check whether the frontier has appropriate size

From the data:

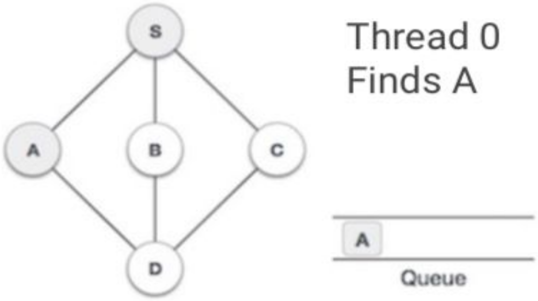
➔ When there are 32768 nodes, the optimized Alpha is at about 16.

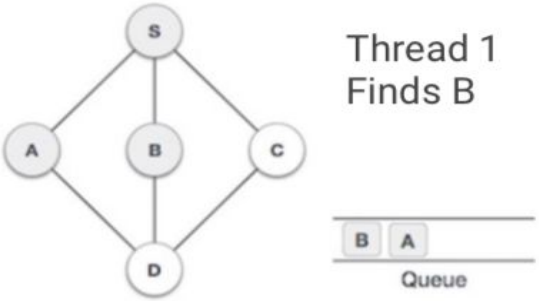
Usage: Compensate for up-bottom finishing

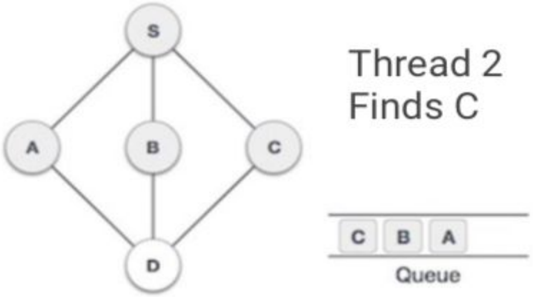


Parallelization

Serial Method with Parallelization: Threads







We see an adjacent node from S, which is A, B, C. For example, we find A and enqueue it.

Then, we enqueue and mark visited B that is adjacent from S.

Finally, the next adjacent node C is visited and enqueue.

Serial Method with Parallelization: Threads

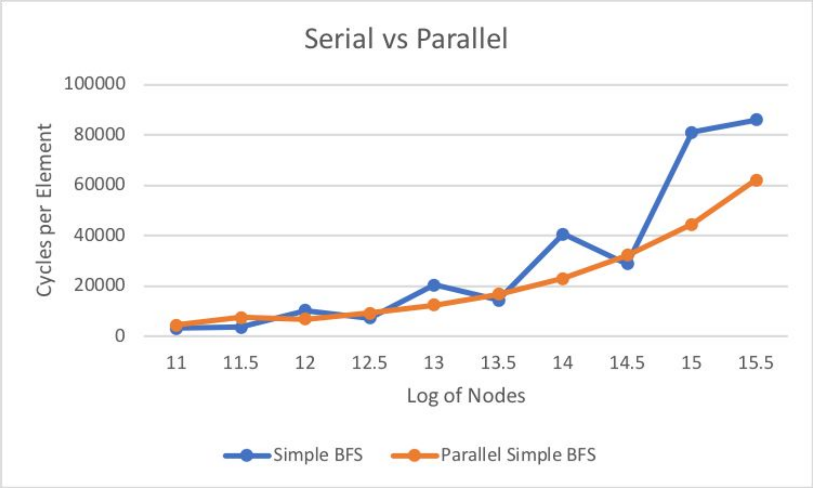
- ❖ The head of the queue is shared to all threads
- ❖ The threads will split into several neighbors to visit
- ❖ The threads will wait for the next update of the queue; otherwise, it stays in rest
- ❖ Queue is locked

Data Comparison

Overall, the performance of parallelization improves. However, the serial implementation is quite unstable, which sometimes performs better than the parallel implementation.

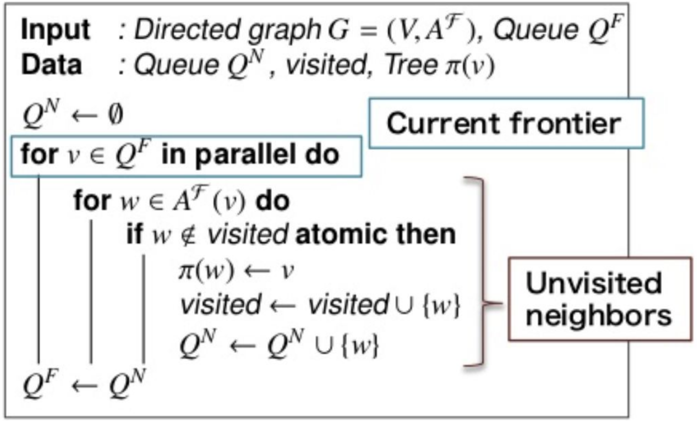
Reason:

It sometimes spends more time waiting.



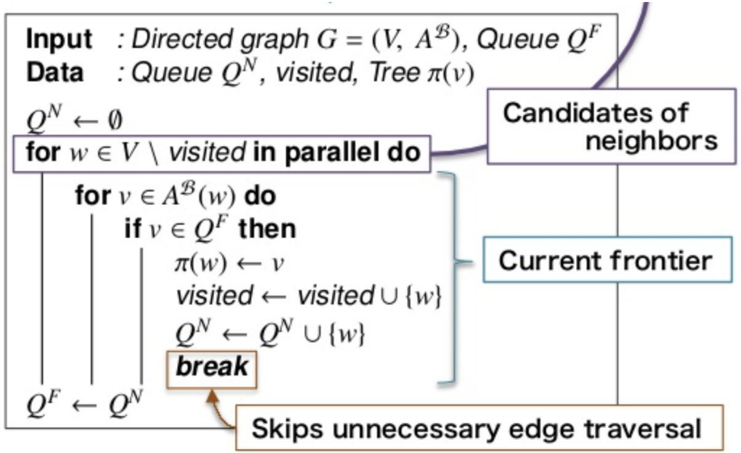
Hybrid Algorithm with Parallelization

Top-Bottom



```
Input : Directed graph $G = (V, A^{\mathcal{F}})$, Queue $Q^F$
Data  : Queue $Q^N$, visited, Tree $\pi(v)$
$Q^N \leftarrow \emptyset$
for $v \in Q^F$ in parallel do
    for $w \in A^{\mathcal{F}}(v)$ do
        if $w \notin$ visited atomic then
            $\pi(w) \leftarrow v$
            visited $\leftarrow$ visited $\cup \{w\}$
            $Q^N \leftarrow Q^N \cup \{w\}$
$Q^F \leftarrow Q^N$
```

Bottom-Top



```
Input : Directed graph $G = (V, A^{\mathcal{B}})$, Queue $Q^F$
Data  : Queue $Q^N$, visited, Tree $\pi(v)$
$Q^N \leftarrow \emptyset$
for $w \in V \setminus$ visited in parallel do
    for $v \in A^{\mathcal{B}}(w)$ do
        if $v \in Q^F$ then
            $\pi(w) \leftarrow v$
            visited $\leftarrow$ visited $\cup \{w\}$
            $Q^N \leftarrow Q^N \cup \{w\}$
            break
$Q^F \leftarrow Q^N$
```

Make good use of barriers: each level of nodes

Image Source: https://www.slideshare.net/yuichiroyasui/research-paper-08yasuiisc14revised

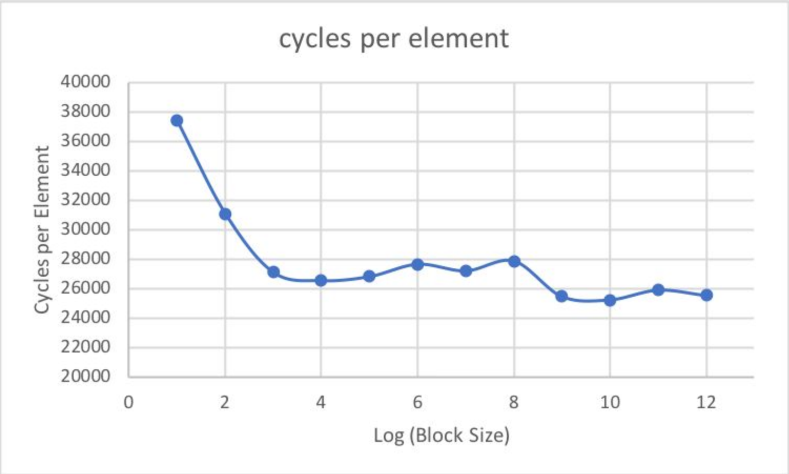
Blocked Parallel Hybrid

- ➔ Temporal Locality

From the data:

- ➔ We found the optimized blocked size is 1024 for 32768 nodes.

Overall, as blocked size increase, the number of iterations decreases.



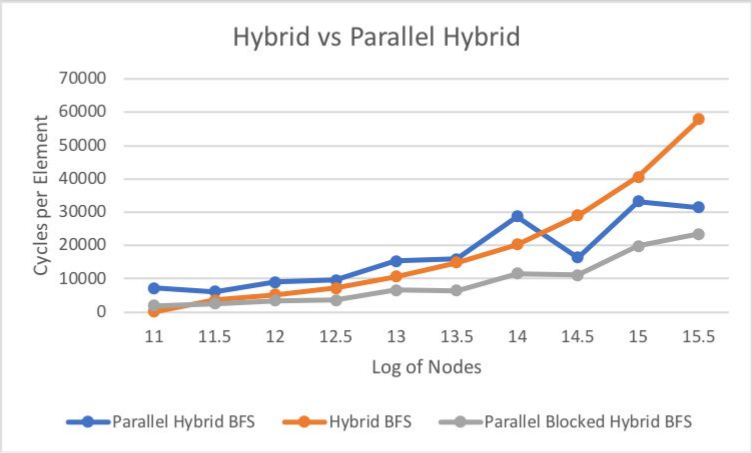
Data Comparison

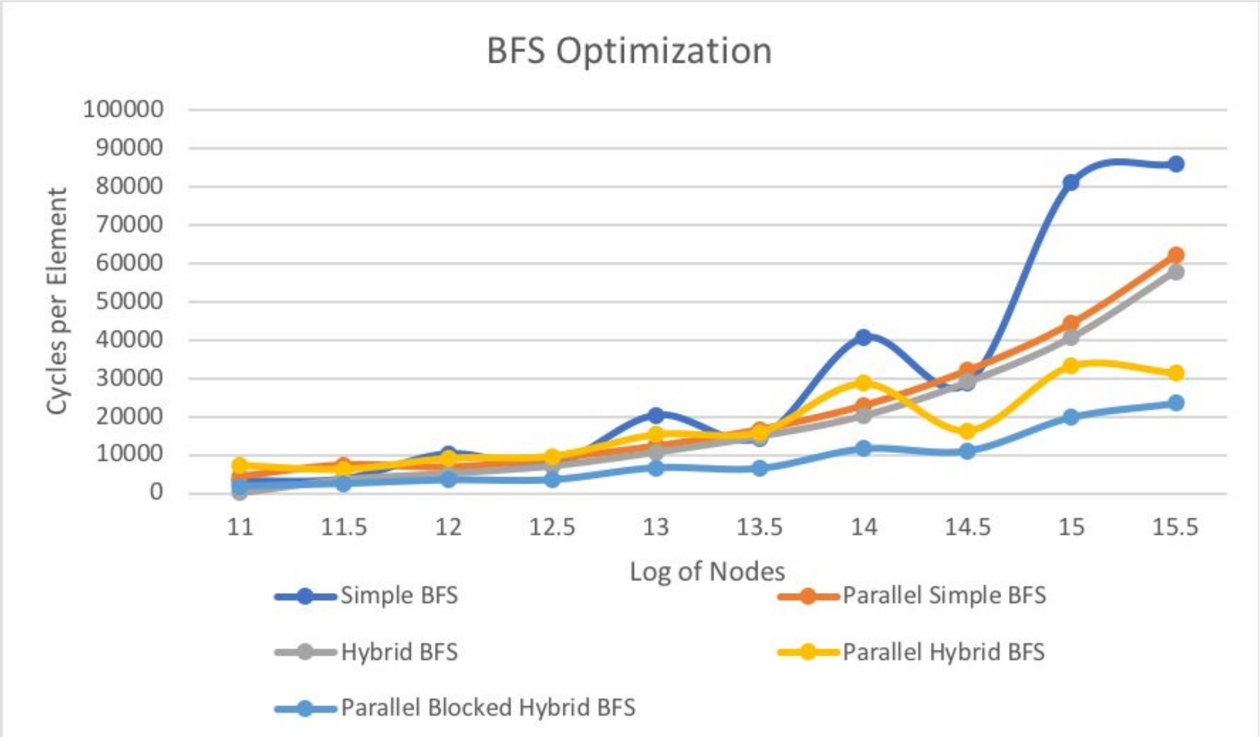
By comparing serial hybrid, parallel hybrid, and parallel blocked hybrid:

When there are large number of nodes:

- ❖ Multi Threads are efficient solution
- ❖ Parallel steps also helps

Overall, parallel blocked hybrid works the best.





In summary, the efficiency: (Under Large Number of Nodes)

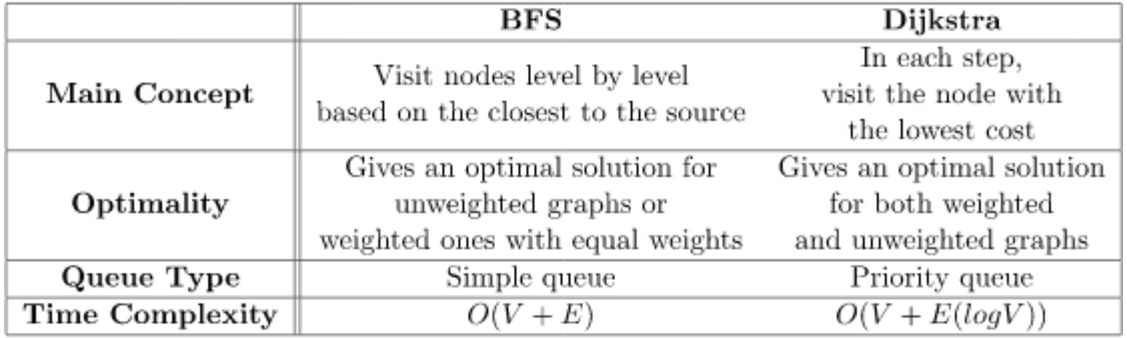
Parallel Blocked Hybrid > Parallel Hybrid > Hybrid > Simple Parallel > Simple

Dijkstra´s Algorithm (GPU)

Definition

Dijkstra's algorithm found the shortest path between two given nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph.

Method



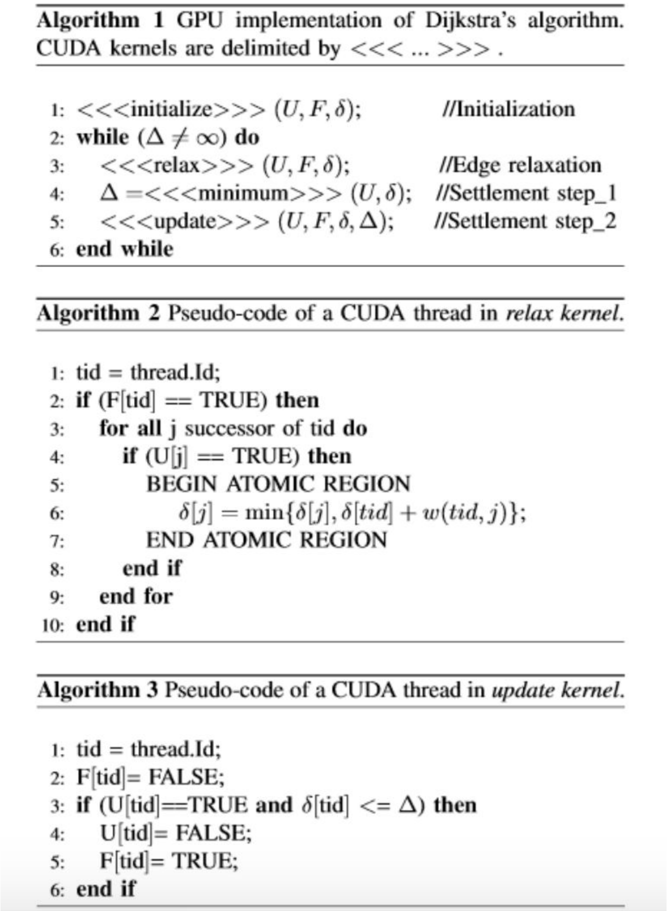
	BFS	Dijkstra
Main Concept	Visit nodes level by level based on the closest to the source	In each step, visit the node with the lowest cost
Optimality	Gives an optimal solution for unweighted graphs or weighted ones with equal weights	Gives an optimal solution for both weighted and unweighted graphs
Queue Type	Simple queue	Priority queue
Time Complexity	$O(V + E)$	$O(V + E(logV))$

- Breadth-first search is just Dijkstra's algorithm with all edge weights equal to 1.
- Kernel call to initialize all node weights to infinity except for the source node. We mark the source node as settled after this point.
- Kernel Call to choose a node which is relaxed and settled and to relax the outgoing edges of each settled node.
- Kernel call to mark all the settled nodes.

Image Source: https://www.baeldung.com/cs/graph-algorithms-bfs-dijkstra

Optimization

- ❖ Kernel Call to initialize the values of the node weights to infinity except for the Source Node which is set to 0
- ❖ We mark the source Node to be settled after that and all other nodes unsettled
- ❖ Kernel Call to choose the node with minimum node weight whose edges are to be relaxed
- ❖ Kernel call to relax all the edges of a node



Algorithm 1 GPU implementation of Dijkstra's algorithm. CUDA kernels are delimited by <<< ... >>> .

```
<<<initialize>>> $(U, F, \delta)$;              //Initialization
while ($\Delta \neq \infty$) do
    <<<relax>>> $(U, F, \delta)$;               //Edge relaxation
    $\Delta$ =<<<minimum>>> $(U, \delta)$;      //Settlement step_1
    <<<update>>> $(U, F, \delta, \Delta)$;      //Settlement step_2
end while
```

Algorithm 2 Pseudo-code of a CUDA thread in *relax kernel*.

```
tid = thread.Id;
if (F[tid] == TRUE) then
    for all j successor of tid do
        if (U[j] == TRUE) then
            BEGIN ATOMIC REGION
                $\delta[j] = \min\{\delta[j], \delta[tid] + w(tid, j)\}$;
            END ATOMIC REGION
        end if
    end for
end if
```

Algorithm 3 Pseudo-code of a CUDA thread in *update kernel*.

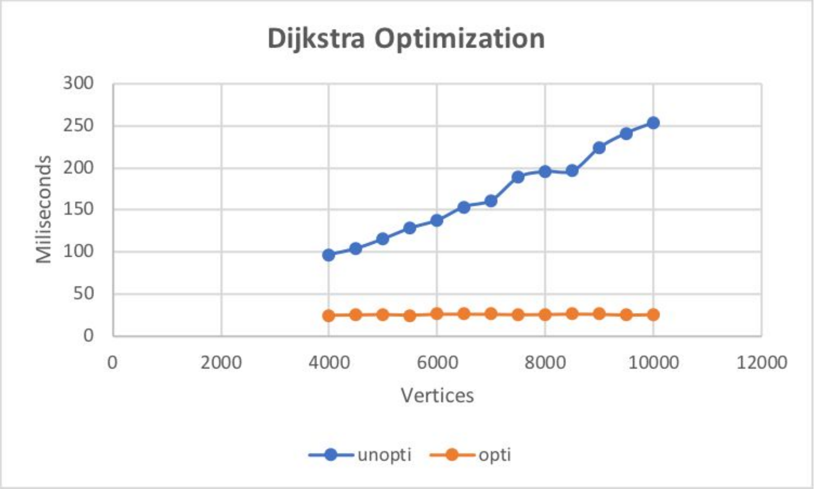
```
tid = thread.Id;
F[tid]= FALSE;
if (U[tid]==TRUE and $\delta$[tid] <= $\Delta$) then
    U[tid]= FALSE;
    F[tid]= TRUE;
end if
```

Image Source: http://ise.ait.ac.th/wp-content/uploads/sites/57/2020/12/Parallel-Shortest-Path-Algorithms-for-Graphics-Processing-Units.pdf

Data Comparison

Under Edge per Nodes = 4000 & Weight of each Edge = 5:

After Optimization, the number of vertices does not seem to have influence on the runtime.



Implementation on GPU

Serial vs Parallel (GPU)

With 1024 threads, 10000000 vertices, 50000000 edges

Sequential Execution Time: 3.81 sec

Kernel Execution Time: 0.02636 sec

Speed up by: 144.554 times

Thank you