CS6068
Parallel Computing - Fall 2014
Lecture Week 6 – Oct 13

Topics:

Unit 6 Udacity Continues

Review Breadth-First Graph Traversals

Depth-first and P-complete problems

Processing Linked Lists in Parallel

Trading Work for Steps

Bloom Filters and Cuckoo Hashing

Parallel Graph Traversal

WWW, Facebook, Tor

Application: Visit every node once

Breadth-First Traversal:

Visit nodes level-by-level, synchronously

Depth-First Traversal:

Visit nodes at periphery first.

Understand Variety of Graphs : Small Depth, Large Depth, Small-world

Design of BFS

Goal: Compute hop distance of every node from source node.

1st try:

Thread per Edge Method

Work Complexity

Step Complexity

Control Iterations

Race Conditions

Finishing Conditions

INITIALIZE: STARTING NODE'S DEPTH = 0

STARTING FRONTIER = NEIGHBORS OF STARTING NODE

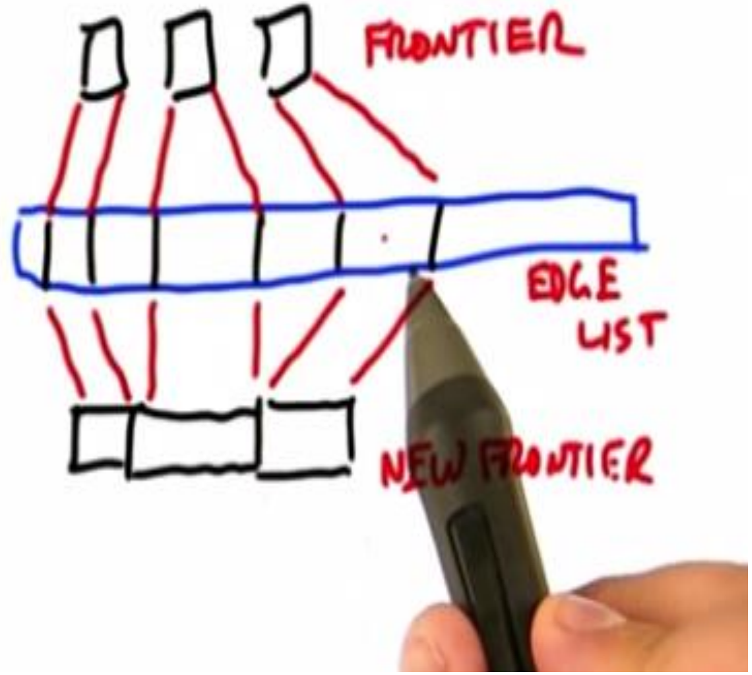
1 FRONTIER: FIND NEIGHBOR START

2 : HOW MANY NEIGHBORS?

3 ALLOCATE SPACE FOR NEW FRONTIER

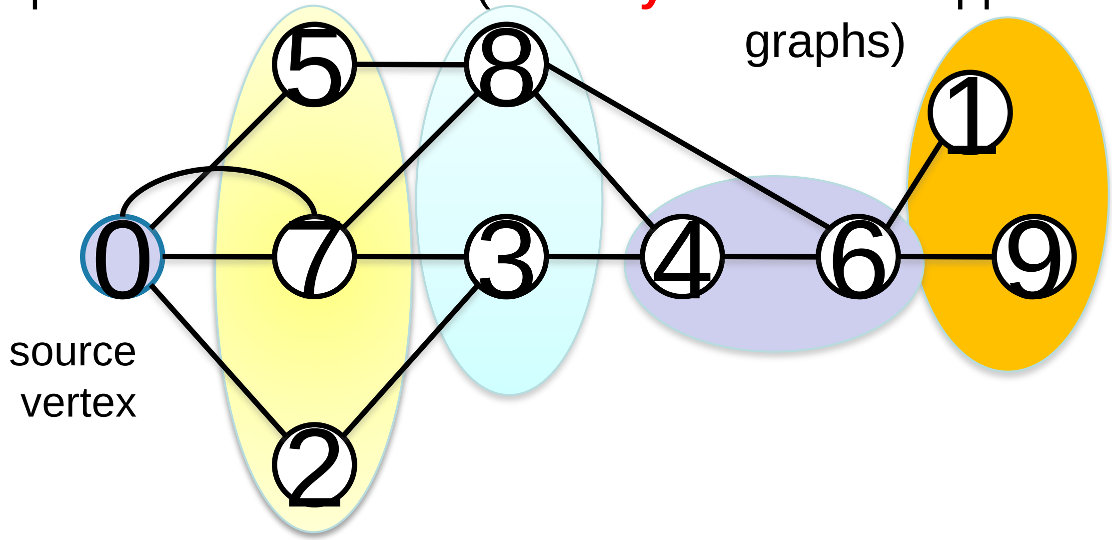
4 COPY EDGE LIST TO NEW ARRAY

5 CULL VISITED ELEMENTS



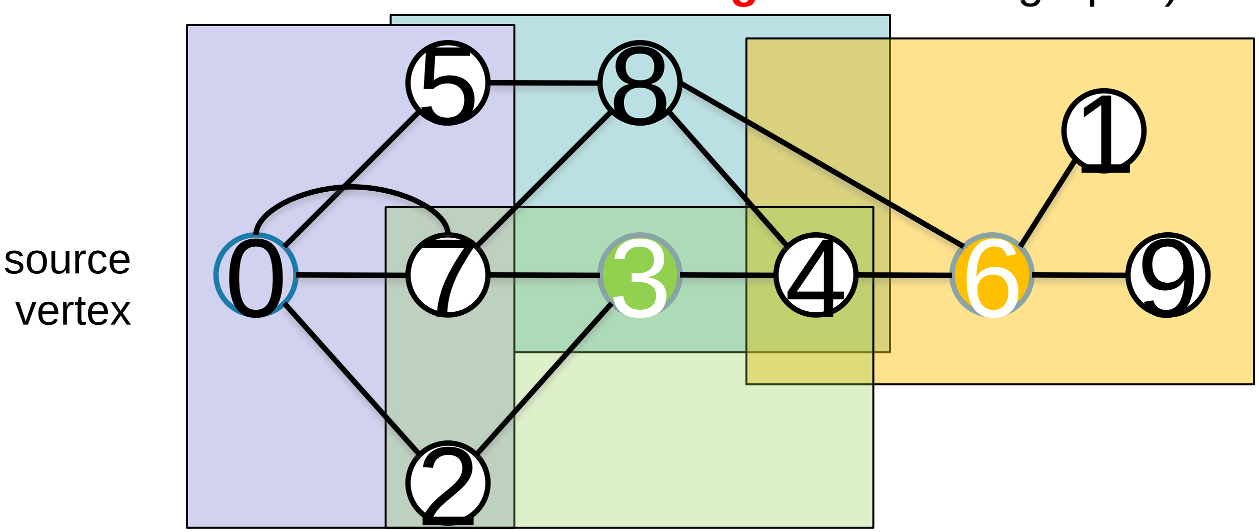
Parallel BFS Strategies

1. Expand current frontier (**level-synchronous** approach, suited for **low diameter** graphs)



- O(D) parallel steps
- Adjacencies of all vertices in current frontier are visited in parallel

2. Stitch multiple concurrent traversals (Ullman-Yannakakis approach, suited for **high-diameter** graphs)



- path-limited searches from “super vertices”
- APSP between “super vertices”

Depth-first Traversal

```
def depthFirst (v):
  if marked[v]: return
  else:
     marked[v]=true
     for each w in Neighbor(v):
          depthFirst(w)
```

Depth-first traversal is used in many apps

-Topological sort of DAGs

-Detecting cycles in graphs

-Strongly-connected components

Depth-first ordering is labeling of vertices that are consistent with the ordering of a depth-first traversal.

The problem of computing depth-first ordering is P-complete.

Parallel Complexity Theory

What problems are inherently sequential?

Recall the sequential complextiy classes:

P, NP, NP-complete 1MQ: P = NP ?

A class to describe parallel problems: NC

- solvable in polylog time on poly number of processors

Now we can ask what are the hardest polytime sequential problems? Call this **P-complete**

Examples:

Circuit Value Problem

Conway’s Game of Life

Lexicographically First Depth First Ordering

Processing a Linked List

Design Strategy: Recursive Pointer Jumping

The idea is to do pointer indirection in parallel and turn a single linked list into 2 shorter (half the length) linked lists in one operation, then recurse.

Here is the basic code:

```
def pointer_jump_rec (list-of-ptrs):
   chum-ptrs = range(len(list-of-ptrs))
   for i in range(len(chum-ptrs)):
       chumptr[i] = list-of-ptrs[list-of-ptrs[i]]
   if eql(chum-ptrs, list-of-ptrs):
        return chum-ptrs
   else:
     return pointer_jump_rec(chum-ptrs)
```

An Example of using Pointer Jumping

0-> 2->1-> 5-> 6   4-> 7-> 3

we have two linked lists that can be stored as one python list of index pointers

ptrs=[2, 5, 1, 3, 7, 6, 6, 3].

If we apply pointer jumping each node index will eventually point to the head of the list it is a member of.

```
>>> pointer_jump_rec ([2, 5, 1, 3, 7, 6, 6, 3])
[6, 6, 6, 3, 3, 6, 6, 3]
```

Work and Step complexity of Pointer-Jumping??

List Ranking Problem:

Determine the hop distance of each node to its head node.

Can we apply Pointer-Jumping paradigm to solve List-Ranking Problem?

We can update the rank of node with each call to pointer-jump:

```
def jump_rank (ptrs, oldranks):
  newranks= range(len(oldranks))
  for i in range(len(ranks)):
       newranks[i] = oldranks[i] + oldranks[ptrs[i]]
  chum-ptrs = range(len(ptrs))
  for i in range(len(ptrs)):
       chum-ptrs[i] = ptrs[ptrs[i]]
  if eql(chumptrs,ptrs):
       return newranks
  else:
       return jump_rank(chum-ptrs,newranks)
```

Why does this work? What are initial ranks?

What is the invariant property of (chum-ptrs,newranks)

Trading-off Work for Steps

List ranking is a classic example that shows that we can trade-off increasing work for reduction in step complexity.

Let's apply this principle to sorting. Suppose we permit unlimited work. How much can we reduce the step complexity of sorting??

Counting Sort

```
def countingSort(list):
     scatter = range(len(list))
     result = range(len(list))
     for i in range(len(list)): #do for loop in parallel
          scatter[i] = len(compact(lessthan(list[i]), list))
          result[scatter[i]] = list[i]
    return result
```

What is the work and step complexity of countingSort??

Parallel Hashing

Hashing Problem: Fast lookup in table using keys.

Applications: Implementing Disjoint Sets, Graph lookups

Static Case: Perfect Hashing

Dynamic: Insertions and deletions, bucket contention and collisions

Load Factors – if your hashing n items to m buckets what is optimal load factor?

Separate Chaining

Open Addressing

Problems with Chaining and Open Addressing in parallel.

Bloom Filters for Set Membership

Problem: Fast test of inclusion – is given element

already in a given set.

Bloom Filter BF Solution:

Select k-hash functions map to locations in BF table – large table of bits – initially all 0

Insert item into set by setting k-bits in BF using k-hashes

Testing for set inclusion checks k-bits

Small probability of false positives.

Cuckoo Hashing – avoid problems with chaining

Select k>1 hash functions

To insert item - Map item to any of the k locations

If there are k-collisions we bump one from nest.

Reinsert the bumped element.

When can this fail? Under what conditions?

Is it likely to fail? How large do we choose n?

k	h(k)	h'(k)
20	9	1
50	6	4
53	9	4
75	9	6
100	1	9
67	1	6
105	6	9
3	3	0
36	3	3
39	6	3

1. table for h(k)										
	20	50	53	75	100	67	105	3	36	39
0										
1					100	67	67	67	67	100
2										
3								3	3	36
4										
5										
6		50	50	50	50	50	105	105	105	50
7										
8										
9	20	20	20	20	20	20	53	53	53	75
10										

2. table for h'(k)										
	20	50	53	75	100	67	105	3	36	39
0										3
1							20	20	20	20
2										
3									36	39
4			53	53	53	53	50	50	50	53
5										
6				75	75	75	75	75	75	67
7										
8										
9						100	100	100	100	105
10										